Stationary variational forms

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

2016

PRELIMINARY VERSION

Contents

1 Basic principles for approximating differential equations

The finite element method is a very flexible approach for solving partial differential equations. Its two most attractive features are the ease of handling domains of complex shape in two and three dimensions and the ease of using higher-degree

polynomials in the approximations. The latter feature typically leads to errors proportional to h^{d+1}, where h is the element length and d is the polynomial degree. When the solution is sufficiently smooth, the ability to use larger d creates methods that are much more computationally efficient than standard finite difference methods (and equally efficient finite difference methods are technically much harder to construct).

The finite element method is usually applied for discretization in space, and therefore spatial problems will be our focus in the coming sections. Extensions to time-dependent problems usually employs finite difference approximations in time.

Before studying how finite element methods are used to tackle differential equations, we first look at how global basis functions and the least squares, Galerkin, and collocation principles can be used to solve differential equations.

1.1 Differential equation models

Let us consider an abstract differential equation for a function $u(x)$ of one variable, written as

$$\mathcal{L}(u) = 0, \quad x \in \Omega\,. \tag{1}$$

Here are a few examples on possible choices of $\mathcal{L}(u)$, of increasing complexity:

$$\mathcal{L}(u) = \frac{d^2u}{dx^2} - f(x), \tag{2}$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(\alpha(x)\frac{du}{dx}\right) + f(x), \tag{3}$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(\alpha(u)\frac{du}{dx}\right) - au + f(x), \tag{4}$$

$$\mathcal{L}(u) = \frac{d}{dx}\left(\alpha(u)\frac{du}{dx}\right) + f(u,x)\,. \tag{5}$$

Both $\alpha(x)$ and $f(x)$ are considered as specified functions, while a is a prescribed parameter. Differential equations corresponding to (2)-(3) arise in diffusion phenomena, such as stationary (time-independent) transport of heat in solids and flow of viscous fluids between flat plates. The form (4) arises when transient diffusion or wave phenomena are discretized in time by finite differences. The equation (5) appears in chemical models when diffusion of a substance is combined with chemical reactions. Also in biology, (5) plays an important role, both for spreading of species and in models involving generation and propagation of electrical signals.

Let $\Omega = [0, L]$ be the domain in one space dimension. In addition to the differential equation, u must fulfill boundary conditions at the boundaries of the domain, $x = 0$ and $x = L$. When $\mathcal{L}$ contains up to second-order derivatives, as in the examples above, we need one boundary condition at each of the (two) boundary points, here abstractly specified as

$$\mathcal{B}_0(u) = 0,\ x = 0, \quad \mathcal{B}_1(u) = 0,\ x = L \tag{6}$$

There are three common choices of boundary conditions:

$$\mathcal{B}_i(u) = u - g, \qquad \text{Dirichlet condition} \tag{7}$$

$$\mathcal{B}_i(u) = -\alpha\frac{du}{dx} - g, \qquad \text{Neumann condition} \tag{8}$$

$$\mathcal{B}_i(u) = -\alpha\frac{du}{dx} - H(u - g), \qquad \text{Robin condition} \tag{9}$$

Here, g and H are specified quantities.

From now on we shall use $u_{\mathrm{e}}(x)$ as symbol for the *exact* solution, fulfilling

$$\mathcal{L}(u_{\mathrm{e}}) = 0, \quad x \in \Omega, \tag{10}$$

while $u(x)$ is our notation for an *approximate* solution of the differential equation.

Remark on notation.

In the literature about the finite element method, it is common to use u as the exact solution and u_h as the approximate solution, where h is a discretization parameter. However, the vast part of the present text is about the approximate solutions, and having a subscript h attached all the time is cumbersome. Of equal importance is the close correspondence between implementation and mathematics that we strive to achieve in this text: when it is natural to use `u` and not `u_h` in code, we let the mathematical notation be dictated by the code's preferred notation. In the relatively few cases where we need to work with the exact solution of the PDE problem we call it u_{e} in mathematics and `u_e` in the code (the function for computing `u_e` is named `u_exact`).

1.2 Simple model problems and their solutions

A common model problem used much in the forthcoming examples is

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = 0,\ u(L) = D\,. \tag{11}$$

A closely related problem with a different boundary condition at $x = 0$ reads

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u'(0) = C,\ u(L) = D\,. \tag{12}$$

A third variant has a variable coefficient,

$$-(\alpha(x)u'(x))' = f(x), \quad x \in \Omega = [0, L], \quad u'(0) = C,\ u(L) = D\,. \tag{13}$$

We can easily solve these model problems using `sympy`. Some common code is defined first:

```
import sympy as sym
x, L, C, D, c_0, c_1, = sym.symbols('x L C D c_0 c_1')
```

The following function computes the solution symbolically for the model problem (11):

```
def model1(f, L, D):
    """Solve -u'' = f(x), u(0)=0, u(L)=D."""
    # Integrate twice
    u_x = - sym.integrate(f, (x, 0, x)) + c_0
    u = sym.integrate(u_x, (x, 0, x)) + c_1
    # Set up 2 equations from the 2 boundary conditions and solve
    # with respect to the integration constants c_0, c_1
    r = sym.solve([u.subs(x, 0)-0,  # x=0 condition
                   u.subs(x,L)-D],  # x=L condition
                  [c_0, c_1])       # unknowns
    # Substitute the integration constants in the solution
    u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
    u = sym.simplify(sym.expand(u))
    return u
```

Calling `model1(2, L, D)` results in the solution

$$u(x) = \frac{1}{L}x\left(D + L^2 - Lx\right) \tag{14}$$

The model problem (12) can be solved by

```
def model2(f, L, C, D):
    """Solve -u'' = f(x), u'(0)=C, u(L)=D."""
    u_x = - sym.integrate(f, (x, 0, x)) + c_0
    u = sym.integrate(u_x, (x, 0, x)) + c_1
    r = sym.solve([sym.diff(u,x).subs(x, 0)-C,  # x=0 cond.
                   u.subs(x,L)-D],              # x=L cond.
                  [c_0, c_1])
    u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
    u = sym.simplify(sym.expand(u))
    return u
```

to yield

$$u(x) = -x^2 + Cx - CL + D + L^2, \tag{15}$$

if $f(x) = 2$. Model (13) requires a bit more involved code,

```
def model3(f, a, L, C, D):
    """Solve -(a*u')' = f(x), u(0)=C, u(L)=D."""
    au_x = - sym.integrate(f, (x, 0, x)) + c_0
    u = sym.integrate(au_x/a, (x, 0, x)) + c_1
    r = sym.solve([u.subs(x, 0)-C,
                   u.subs(x,L)-D],
                  [c_0, c_1])
    u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
    u = sym.simplify(sym.expand(u))
```

```
    return u


def demo():
    f = 2
    u = model1(f, L, D)
    print 'model1:', u, u.subs(x, 0), u.subs(x, L)
    print sym.latex(u, mode='plain')
    u = model2(f, L, C, D)
    #f = x
    #u = model2(f, L, C, D)
    print 'model2:', u, sym.diff(u, x).subs(x, 0), u.subs(x, L)
    print sym.latex(u, mode='plain')
    u = model3(0, 1+x**2, L, C, D)
    print 'model3:', u, u.subs(x, 0), u.subs(x, L)
    print sym.latex(u, mode='plain')

if __name__ == '__main__':
    demo()
```

With $f(x) = 0$ and $\alpha(x) = 1 + x^2$ we get

$$u(x) = \frac{C\tan^{-1}(L) - C\tan^{-1}(x) + D\tan^{-1}(x)}{\tan^{-1}(L)}$$

1.3 Forming the residual

The fundamental idea is to seek an approximate solution u in some space V,

$$V = \text{span}\{\psi_0(x), \ldots, \psi_N(x)\},$$

which means that u can always be expressed as a linear combination of the basis functions $\{\psi_j\}_{j\in\mathcal{I}_s}$, with $\mathcal{I}_s$ as the index set $\{0, \ldots, N\}$:

$$u(x) = \sum_{j\in\mathcal{I}_s} c_j\psi_j(x)\,.$$

The coefficients $\{c_j\}_{j\in\mathcal{I}_s}$ are unknowns to be computed.

(Later, in Section 4, we will see that if we specify boundary values of u different from zero, we must look for an approximate solution $u(x) = B(x) + \sum_j c_j\psi_j(x)$, where $\sum_j c_j\psi_j \in V$ and $B(x)$ is some function for incorporating the right boundary values. Because of $B(x)$, u will not necessarily lie in V. This modification does not imply any difficulties.)

We need principles for deriving $N + 1$ equations to determine the $N + 1$ unknowns $\{c_i\}_{i\in\mathcal{I}_s}$. When approximating a given function f by $u = \sum_j c_j\psi_j$, a key idea is to minimize the square norm of the approximation error $e = u - f$ or (equivalently) demand that e is orthogonal to V. Working with e is not so useful here since the approximation error in our case is $e = u_{\text{e}} - u$ and u_{e} is unknown. The only general indicator we have on the quality of the approximate solution is to what degree u fulfills the differential equation. Inserting $u = \sum_j c_j\psi_j$ into $\mathcal{L}(u)$ reveals that the result is not zero, because u is only likely to equal u_{e}. The nonzero result,

$$R = \mathcal{L}(u) = \mathcal{L}(\sum_j c_j\psi_j), \tag{16}$$

is called the *residual* and measures the error in fulfilling the governing equation.

Various principles for determining $\{c_j\}_{j\in\mathcal{I}_s}$ try to minimize R in some sense. Note that R varies with x and the $\{c_j\}_{j\in\mathcal{I}_s}$ parameters. We may write this dependence explicitly as

$$R = R(x; c_0, \ldots, c_N)\,. \tag{17}$$

Below, we present three principles for making R small: a least squares method, a projection or Galerkin method, and a collocation or interpolation method.

1.4 The least squares method

The least-squares method aims to find $\{c_i\}_{i\in\mathcal{I}_s}$ such that the square norm of the residual

$$||R|| = (R, R) = \int_\Omega R^2\,\mathrm{d}x \tag{18}$$

is minimized. By introducing an inner product of two functions f and g on Ω as

$$(f, g) = \int_\Omega f(x)g(x)\,\mathrm{d}x, \tag{19}$$

the least-squares method can be defined as

$$\min_{c_0,\ldots,c_N} E = (R, R)\,. \tag{20}$$

Differentiating with respect to the free parameters $\{c_i\}_{i\in\mathcal{I}_s}$ gives the $N+1$ equations

$$\int_\Omega 2R\frac{\partial R}{\partial c_i}\,\mathrm{d}x = 0 \quad \Leftrightarrow \quad (R, \frac{\partial R}{\partial c_i}) = 0, \quad i \in \mathcal{I}_s\,. \tag{21}$$

1.5 The Galerkin method

The least-squares principle is equivalent to demanding the error to be orthogonal to the space V when approximating a function f by $u \in V$. With a differential equation we do not know the true error so we must instead require the residual R to be orthogonal to V. This idea implies seeking $\{c_i\}_{i\in\mathcal{I}_s}$ such that

$$(R, v) = 0, \quad \forall v \in V\,. \tag{22}$$

This is the Galerkin method for differential equations.

This statement is equivalent to R being orthogonal to the $N+1$ basis functions individually:

$$(R, \psi_i) = 0, \quad i \in \mathcal{I}_s, \tag{23}$$

resulting in $N+1$ equations for determining $\{c_i\}_{i\in\mathcal{I}_s}$.

1.6 The Method of Weighted Residuals

A generalization of the Galerkin method is to demand that R is orthogonal to some space W, but not necessarily the same space as V where we seek the unknown function. This generalization is called the *method of weighted residuals*:

$$(R, v) = 0, \quad \forall v \in W\,. \tag{24}$$

If $\{w_0, \ldots, w_N\}$ is a basis for W, we can equivalently express the method of weighted residuals as

$$(R, w_i) = 0, \quad i \in \mathcal{I}_s\,. \tag{25}$$

The result is $N+1$ equations for $\{c_i\}_{i\in\mathcal{I}_s}$.

The least-squares method can also be viewed as a weighted residual method with $w_i = \partial R/\partial c_i$.

Variational formulation of the continuous problem.

Statements like (22), (23), (24), or (25)) are known as weak formulations[a] or *variational formulations*. (Other terms are weak and variational forms.) These equations are in this text primarily used for a numerical approximation $u \in V$, where V is a *finite-dimensional* space with dimension $N+1$. However, we may also let the exact solution u_{e} fulfill a variational formulation $(\mathcal{L}(u_{\mathrm{e}}), v) = 0\ \forall v \in V$, but the exact solution lies in general in a space with infinite dimensions (because an infinite number of parameters are needed to specify the solution). The variational formulation for u_{e} in an infinite-dimensional space V is a mathematical way of stating the problem and acts as an alternative to the usual formulation of a differential equation with initial and/or boundary conditions.

Much of the literature on finite element methods takes a differential equation problem and first transforms it to a variational formulation in an infinite-dimensional space V, before searching for an approximate solution in a finite-dimensional subspace of V. However, we prefer the more intuitive approach with an approximate solution u in a finite-dimensional space V inserted in the differential equation, and then the resulting residual is demanded to be orthogonal to V.

[a] `https://en.wikipedia.org/wiki/Weak_formulation`

Remark on terminology.

The terms weak or variational formulations often refer to a statement like (22) or (24) after *integration by parts* has been performed (the integration by parts technique is explained in Section 1.10). The result after integration by parts is what is obtained after taking the *first variation* of a minimization problem (see Section 1.13). However, in this text we use variational formulation as a common term for formulations which, in contrast to the differential equation $R = 0$, instead demand that an average of R is zero: $(R, v) = 0$ for all v in some space.

1.7 Test and Trial Functions

In the context of the Galerkin method and the method of weighted residuals it is common to use the name *trial function* for the approximate $u = \sum_j c_j\psi_j$. The space containing the trial function is known as the *trial space*. The function v entering the orthogonality requirement in the Galerkin method and the method of weighted residuals is called *test function*, and so are the ψ_i or w_i functions that are used as weights in the inner products with the residual. The space where the test functions comes from is naturally called the *test space*.

We see that in the method of weighted residuals the test and trial spaces are different and so are the test and trial functions. In the Galerkin method the test and trial spaces are the same (so far).

1.8 The collocation method

The idea of the collocation method is to demand that R vanishes at $N + 1$ selected points $x_0, \ldots, x_N$ in Ω:

$$R(x_i; c_0, \ldots, c_N) = 0, \quad i \in \mathcal{I}_s\,. \tag{26}$$

The collocation method can also be viewed as a method of weighted residuals with Dirac delta functions as weighting functions. Let $\delta(x - x_i)$ be the Dirac delta function centered around $x = x_i$ with the properties that $\delta(x - x_i) = 0$ for $x \neq x_i$ and

$$\int_\Omega f(x)\delta(x - x_i)\,\mathrm{d}x = f(x_i), \quad x_i \in \Omega\,. \tag{27}$$

Intuitively, we may think of $\delta(x - x_i)$ as a very peak-shaped function around $x = x_i$ with an integral $\int_{-\infty}^{\infty} \delta(x - x_i)dx$ that evaluates to unity. Mathematically, it can be shown that $\delta(x - x_i)$ is the limit of a Gaussian function centered at $x = x_i$ with a standard deviation that approaches zero. Using this latter model, we can roughly visualize delta functions as done in Figure 1. Because of (27), we can let $w_i = \delta(x - x_i)$ be weighting functions in the method of weighted residuals, and (25) becomes equivalent to (26).

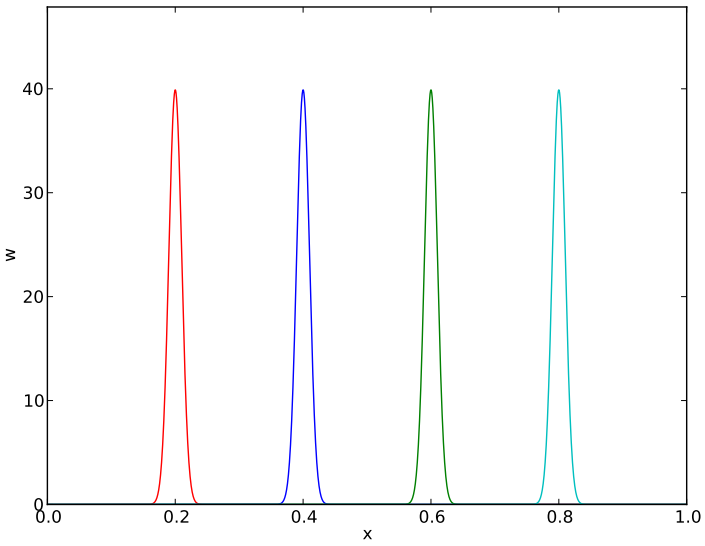


Figure 1: Approximation of delta functions by narrow Gaussian functions.

The subdomain collocation method. The idea of this approach is to demand the integral of R to vanish over $N+1$ subdomains Ω_i of Ω:

$$\int_{\Omega_i} R \, \mathrm{d}x = 0, \quad i \in \mathcal{I}_s \, . \tag{28}$$

This statement can also be expressed as a weighted residual method

$$\int_{\Omega} R w_i \, \mathrm{d}x = 0, \quad i \in \mathcal{I}_s, \tag{29}$$

where $w_i = 1$ for $x \in \Omega_i$ and $w_i = 0$ otherwise.

1.9 Examples on using the principles

Let us now apply global basis functions to illustrate the different principles for making the residual R small.

The model problem. We consider the differential equation problem

$$-u''(x) = f(x), \quad x \in \Omega = [0, L], \quad u(0) = 0, \; u(L) = 0 \, . \tag{30}$$

Basis functions. Our choice of basis functions ψ_i for V is

$$\psi_i(x) = \sin\left((i+1)\pi\frac{x}{L}\right), \quad i \in \mathcal{I}_s\,. \tag{31}$$

An important property of these functions is that $\psi_i(0) = \psi_i(L) = 0$, which means that the boundary conditions on u are fulfilled:

$$u(0) = \sum_j c_j\psi_j(0) = 0, \quad u(L) = \sum_j c_j\psi_j(L) = 0\,.$$

Another nice property is that the chosen sine functions are orthogonal on Ω:

$$\int_0^L \sin\left((i+1)\pi\frac{x}{L}\right)\sin\left((j+1)\pi\frac{x}{L}\right)\,\mathrm{d}x = \begin{cases} \frac{1}{2}L & i = j \\ 0, & i \neq j \end{cases} \tag{32}$$

provided i and j are integers.

The residual. We can readily calculate the following explicit expression for the residual:

$$\begin{aligned} R(x; c_0, \ldots, c_N) &= u''(x) + f(x), \\ &= \frac{d^2}{dx^2}\left(\sum_{j\in\mathcal{I}_s} c_j\psi_j(x)\right) + f(x), \\ &= \sum_{j\in\mathcal{I}_s} c_j\psi_j''(x) + f(x)\,. \end{aligned} \tag{33}$$

The least squares method. The equations (21) in the least squares method require an expression for $\partial R/\partial c_i$. We have

$$\frac{\partial R}{\partial c_i} = \frac{\partial}{\partial c_i}\left(\sum_{j\in\mathcal{I}_s} c_j\psi_j''(x) + f(x)\right) = \sum_{j\in\mathcal{I}_s}\frac{\partial c_j}{\partial c_i}\psi_j''(x) = \psi_i''(x)\,. \tag{34}$$

The governing equations for the unknown parameters $\{c_j\}_{j\in\mathcal{I}_s}$ are then

$$(\sum_j c_j\psi_j'' + f, \psi_i'') = 0, \quad i \in \mathcal{I}_s, \tag{35}$$

which can be rearranged as

$$\sum_{j\in\mathcal{I}_s}(\psi_i'', \psi_j'')c_j = -(f, \psi_i''), \quad i \in \mathcal{I}_s\,. \tag{36}$$

This is nothing but a linear system

$$\sum_{j\in\mathcal{I}_s} A_{i,j}c_j = b_i, \quad i \in \mathcal{I}_s\,.$$

The entries in the coefficient matrix are given by

$$\begin{aligned} A_{i,j} &= (\psi_i'', \psi_j'') \\ &= \pi^4(i+1)^2(j+1)^2L^{-4}\int_0^L \sin\left((i+1)\pi\frac{x}{L}\right)\sin\left((j+1)\pi\frac{x}{L}\right)\,\mathrm{d}x \end{aligned}$$

The orthogonality of the sine functions simplify the coefficient matrix:

$$A_{i,j} = \begin{cases} \frac{1}{2}L^{-3}\pi^4(i+1)^4 & i = j \\ 0, & i \neq j \end{cases} \tag{37}$$

The right-hand side reads

$$b_i = -(f, \psi_i'') = (i+1)^2\pi^2L^{-2}\int_0^L f(x)\sin\left((i+1)\pi\frac{x}{L}\right)\,\mathrm{d}x \tag{38}$$

Since the coefficient matrix is diagonal we can easily solve for

$$c_i = \frac{2L}{\pi^2(i+1)^2}\int_0^L f(x)\sin\left((i+1)\pi\frac{x}{L}\right)\,\mathrm{d}x\,. \tag{39}$$

With the special choice of $f(x) = 2$, the coefficients can be calculated in `sympy` by

```
import sympy as sym

i, j = sym.symbols('i j', integer=True)
x, L = sym.symbols('x L')
f = 2
a = 2*L/(sym.pi**2*(i+1)**2)
c_i = a*sym.integrate(f*sym.sin((i+1)*sym.pi*x/L), (x, 0, L))
c_i = simplify(c_i)
print c_i
```

The answer becomes

$$c_i = 4\frac{L^2\left((-1)^i + 1\right)}{\pi^3\left(i^3 + 3i^2 + 3i + 1\right)}$$

Now, $1 + (-1)^i = 0$ for i odd, so only the coefficients with even index are nonzero. Introducing $i = 2k$ for $k = 0, \ldots, N/2$ to count the relevant indices (for N odd, k goes to $(N-1)/2$), we get the solution

$$u(x) = \sum_{k=0}^{N/2}\frac{8L^2}{\pi^3(2k+1)^3}\sin\left((2k+1)\pi\frac{x}{L}\right)\,. \tag{40}$$

The coefficients decay very fast: $c_2 = c_0/27$, $c_4 = c_0/125$. The solution will therefore be dominated by the first term,

$$u(x) \approx \frac{8L^2}{\pi^3} \sin\left(\pi \frac{x}{L}\right) .$$

The Galerkin method. The Galerkin principle (22) applied to (30) consists of inserting our special residual (33) in (22)

$$(u'' + f, v) = 0, \quad \forall v \in V,$$

or

$$(u'', v) = -(f, v), \quad \forall v \in V . \tag{41}$$

This is the variational formulation, based on the Galerkin principle, of our differential equation. The $\forall v \in V$ requirement is equivalent to demanding the equation $(u'', v) = -(f, v)$ to be fulfilled for all basis functions $v = \psi_i$, $i \in \mathcal{I}_s$, see (22) and (23). We therefore have

$$(\sum_{j \in \mathcal{I}_s} c_j \psi_j'', \psi_i) = -(f, \psi_i), \quad i \in \mathcal{I}_s . \tag{42}$$

This equation can be rearranged to a form that explicitly shows that we get a linear system for the unknowns $\{c_j\}_{j \in \mathcal{I}_s}$:

$$\sum_{j \in \mathcal{I}_s} (\psi_i, \psi_j'') c_j = (f, \psi_i), \quad i \in \mathcal{I}_s . \tag{43}$$

For the particular choice of the basis functions (31) we get in fact the same linear system as in the least squares method because $\psi'' = -(i+1)^2 \pi^2 L^{-2} \psi$. Consequently, the solution $u(x)$ becomes identical to the one produced by the least squares method.

The collocation method. For the collocation method (26) we need to decide upon a set of $N+1$ collocation points in Ω. A simple choice is to use uniformly spaced points: $x_i = i\Delta x$, where $\Delta x = L/N$ in our case ($N \geq 1$). However, these points lead to at least two rows in the matrix consisting of zeros (since $\psi_i(x_0) = 0$ and $\psi_i(x_N) = 0$), thereby making the matrix singular and non-invertible. This forces us to choose some other collocation points, e.g., random points or points uniformly distributed in the interior of Ω. Demanding the residual to vanish at these points leads, in our model problem (30), to the equations

$$-\sum_{j \in \mathcal{I}_s} c_j \psi_j''(x_i) = f(x_i), \quad i \in \mathcal{I}_s, \tag{44}$$

which is seen to be a linear system with entries

$$A_{i,j} = -\psi_j''(x_i) = (j+1)^2 \pi^2 L^{-2} \sin\left((j+1)\pi \frac{x_i}{L}\right),$$

in the coefficient matrix and entries $b_i = 2$ for the right-hand side (when $f(x) = 2$).

The special case of $N = 0$ can sometimes be of interest. A natural choice is then the midpoint $x_0 = L/2$ of the domain, resulting in $A_{0,0} = -\psi_0''(x_0) = \pi^2 L^{-2}$, $f(x_0) = 2$, and hence $c_0 = 2L^2/\pi^2$.

Comparison. In the present model problem, with $f(x) = 2$, the exact solution is $u(x) = x(L - x)$, while for $N = 0$ the Galerkin and least squares method result in $u(x) = 8L^2\pi^{-3}\sin(\pi x/L)$ and the collocation method leads to $u(x) = 2L^2\pi^{-2}\sin(\pi x/L)$. We can quickly use `sympy` to verify that the maximum error occurs at the midpoint $x = L/2$ and find what the errors are. First we set up the error expressions:

```
>>> import sympy as sym
>>> # Computing with Dirichlet conditions: -u''=2 and sines
>>> x, L = sym.symbols('x L')
>>> e_Galerkin = x*(L-x) - 8*L**2*sym.pi**(-3)*sym.sin(sym.pi*x/L)
>>> e_colloc = x*(L-x) - 2*L**2*sym.pi**(-2)*sym.sin(sym.pi*x/L)
```

If the derivative of the errors vanish at $x = L/2$, the errors reach their maximum values here (the errors vanish at the boundary points).

```
>>> dedx_Galerkin = sym.diff(e_Galerkin, x)
>>> dedx_Galerkin.subs(x, L/2)
0
>>> dedx_colloc = sym.diff(e_colloc, x)
>>> dedx_colloc.subs(x, L/2)
0
```

Finally, we can compute the maximum error at $x = L/2$ and evaluate the expressions numerically with three decimals:

```
>>> sym.simplify(e_Galerkin.subs(x, L/2).evalf(n=3))
-0.00812*L**2
>>> sym.simplify(e_colloc.subs(x, L/2).evalf(n=3))
0.0473*L**2
```

The error in the collocation method is about 6 times larger than the error in the Galerkin or least squares method.

1.10 Integration by parts

A problem arises if we want to apply popular finite element functions to solve our model problem (30) by the standard least squares, Galerkin, or collocation methods: the piecewise polynomials $\psi_i(x)$ have discontinuous derivatives at the cell boundaries which makes it problematic to compute the second-order derivative. This fact actually makes the least squares and collocation methods less suitable for finite element approximation of the unknown function. (By rewriting the equation $-u'' = f$ as a system of two first-order equations, $u' = v$ and $-v' = f$, the least squares method can be applied. Also, differentiating discontinuous

functions can actually be handled by distribution theory in mathematics.) The Galerkin method and the method of weighted residuals can, however, be applied together with finite element basis functions if we use *integration by parts* as a means for transforming a second-order derivative to a first-order one.

Consider the model problem (30) and its Galerkin formulation

$$-(u'', v) = (f, v) \quad \forall v \in V\,.$$

Using integration by parts in the Galerkin method, we can "move" a derivative of u onto v:

$$\begin{aligned}\int_0^L u''(x)v(x)\,\mathrm{d}x &= -\int_0^L u'(x)v'(x)\,\mathrm{d}x + [vu']_0^L \\ &= -\int_0^L u'(x)v'(x)\,\mathrm{d}x + u'(L)v(L) - u'(0)v(0)\,. \end{aligned} \tag{45}$$

Usually, one integrates the problem at the stage where the u and v functions enter the formulation. Alternatively, but less common, we can integrate by parts in the expressions for the matrix entries:

$$\begin{aligned}\int_0^L \psi_i(x)\psi_j''(x)\,\mathrm{d}x &= -\int_0^L \psi_i'(x)\psi_j'(x)dx + [\psi_i\psi_j']_0^L \\ &= -\int_0^L \psi_i'(x)\psi_j'(x)\,\mathrm{d}x + \psi_i(L)\psi_j'(L) - \psi_i(0)\psi_j'(0)\,. \end{aligned} \tag{46}$$

Integration by parts serves to reduce the order of the derivatives and to make the coefficient matrix symmetric since $(\psi_i', \psi_j') = (\psi_i', \psi_j')$. The symmetry property depends on the type of terms that enter the differential equation. As will be seen later in Section 5, integration by parts also provides a method for implementing boundary conditions involving u'.

With the choice (31) of basis functions we see that the "boundary terms" $\psi_i(L)\psi_j'(L)$ and $\psi_i(0)\psi_j'(0)$ vanish since $\psi_i(0) = \psi_i(L) = 0$. We therefore end up with the following alternative Galerkin formulation:

$$-(u'', v) = (u', v') = (f, v) \quad \forall v \in V\,.$$

Weak form. Since the variational formulation after integration by parts makes weaker demands on the differentiability of u and the basis functions ψ_i, the resulting integral formulation is referred to as a *weak form* of the differential equation problem. The original variational formulation with second-order derivatives, or the differential equation problem with second-order derivative, is then the *strong form*, with stronger requirements on the differentiability of the functions.

For differential equations with second-order derivatives, expressed as variational formulations and solved by finite element methods, we will always perform integration by parts to arrive at expressions involving only first-order derivatives.

1.11 Boundary function

So far we have assumed zero Dirichlet boundary conditions, typically $u(0) = u(L) = 0$, and we have demanded that $\psi_i(0) = \psi_i(L) = 0$ for $i \in \mathcal{I}_s$. What about a boundary condition like $u(L) = D \neq 0$? This condition immediately faces a problem: $u = \sum_j c_j \varphi_j(L) = 0$ since all $\varphi_i(L) = 0$.

A boundary condition of the form $u(L) = D$ can be implemented by demanding that all $\psi_i(L) = 0$, but adding a *boundary function* $B(x)$ with the right boundary value, $B(L) = D$, to the expansion for u:

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)\,.$$

This u gets the right value at $x = L$:

$$u(L) = B(L) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(L) = B(L) = D\,.$$

The idea is that for any boundary where u is known we demand ψ_i to vanish and construct a function $B(x)$ to attain the boundary value of u. There are no restrictions on how $B(x)$ varies with x in the interior of the domain, so this variation needs to be constructed in some way. Exactly how we decide the variation to be, is not important.

For example, with $u(0) = 0$ and $u(L) = D$, we can choose $B(x) = xD/L$, since this form ensures that $B(x)$ fulfills the boundary conditions: $B(0) = 0$ and $B(L) = D$. The unknown function is then sought on the form

$$u(x) = \frac{x}{L} D + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x), \tag{47}$$

with $\psi_i(0) = \psi_i(L) = 0$.

The particular shape of the $B(x)$ function is not important as long as its boundary values are correct. For example, $B(x) = D(x/L)^p$ for any power p will work fine in the above example. Another choice could be $B(x) = D\sin(\pi x/(2L))$.

As a more general example, consider a domain $\Omega = [a, b]$ where the boundary conditions are $u(a) = U_a$ and $u(b) = U_b$. A class of possible $B(x)$ functions is

$$B(x) = U_a + \frac{U_b - U_a}{(b-a)^p}(x-a)^p, \quad p > 0\,. \tag{48}$$

Real applications will most likely use the simplest version, $p = 1$, but here such a p parameter was included to demonstrate that there are many choices of $B(x)$ in a problem. Fortunately, there is a general, unique technique for constructing $B(x)$ when we use finite element basis functions for V.

How to deal with nonzero Dirichlet conditions.

The general procedure of incorporating Dirichlet boundary conditions goes as follows. Let $\partial\Omega_E$ be the part(s) of the boundary $\partial\Omega$ of the domain Ω where u is specified. Set $\psi_i = 0$ at the points in $\partial\Omega_E$ and seek u as

$$u(x) = B(x) + \sum_{j\in\mathcal{I}_s} c_j\psi_j(x), \tag{49}$$

where $B(x)$ equals the boundary conditions on u at $\partial\Omega_E$.

Remark. With the $B(x)$ term, u does not in general lie in $V = \text{span}\,\{\psi_0, \ldots, \psi_N\}$ anymore. Moreover, when a prescribed value of u at the boundary, say $u(a) = U_a$ is different from zero, it does not make sense to say that u lies in a vector space, because this space does not obey the requirements of addition and scalar multiplication. For example, $2u$ does not lie in the space since its boundary value is $2U_a$, which is incorrect. It only makes sense to split u in two parts, as done above, and have the unknown part $\sum_j c_j\psi_j$ in a proper function space.

1.12 Abstract notation for variational formulations

We have seen that variational formulations end up with a formula involving u and v, such as (u', v') and a formula involving v and known functions, such as (f, v). A widely used notation is to introduce an abstract variational statement written as

$$a(u, v) = L(v) \quad \forall v \in V,$$

where $a(u, v)$ is a so-called *bilinear form* involving all the terms that contain both the test and trial function, while $L(v)$ is a *linear form* containing all the terms without the trial function. For example, the statement

$$\int_\Omega u'v'\,\mathrm{d}x = \int_\Omega fv\,\mathrm{d}x \quad \text{or} \quad (u', v') = (f, v) \quad \forall v \in V$$

can be written in abstract form: *find u such that*

$$a(u, v) = L(v) \quad \forall v \in V,$$

where we have the definitions

$$a(u, v) = (u', v'), \quad L(v) = (f, v)\,.$$

The term *linear* means that

$$L(\alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 L(v_1) + \alpha_2 L(v_2)$$

for two test functions v_1 and v_2, and scalar parameters α_1 and α_2. Similarly, the term *bilinear* means that $a(u, v)$ is linear in both its arguments:

$$a(\alpha_1 u_1 + \alpha_2 u_2, v) = \alpha_1 a(u_1, v) + \alpha_2 a(u_2, v),$$
$$a(u, \alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 a(u, v_1) + \alpha_2 a(u, v_2)\,.$$

In nonlinear problems these linearity properties do not hold in general and the abstract notation is then

$$F(u; v) = 0 \quad \forall v \in V\,.$$

The matrix system associated with $a(u, v) = L(v)$ can also be written in an abstract form by inserting $v = \psi_i$ and $u = \sum_j c_j \psi_j$ in $a(u, v) = L(v)$. Using the linear properties, we get

$$\sum_{j \in \mathcal{I}_s} a(\psi_j, \psi_i) c_j = L(\psi_i), \quad i \in \mathcal{I}_s,$$

which is a linear system

$$\sum_{j \in \mathcal{I}_s} A_{i,j} c_j = b_i, \quad i \in \mathcal{I}_s,$$

where

$$A_{i,j} = a(\psi_j, \psi_i), \quad b_i = L(\psi_i)\,.$$

In many problems, $a(u, v)$ is symmetric such that $a(\psi_j, \psi_i) = a(\psi_i, \psi_j)$. In those cases the coefficient matrix becomes symmetric, $A_{i,j} = A_{j,i}$, a property that can simplify solution algorithms for linear systems and make them more stable. The property also reduces memory requirements and the computational work.

The abstract notation $a(u, v) = L(v)$ for linear differential equation problems is much used in the literature and in description of finite element software (in particular the FEniCS[1] documentation). We shall frequently summarize variational forms using this notation.

1.13 Variational problems and minimization of functionals

Example. Many physical problems can be modeled as partial differential equation and as a minimization problem. For example, the deflection $u(x)$ of an elastic string subject to a transversal force $f(x)$ is governed by the differential equation problem

$$-u''(x) = f(x), \quad x \in (0, L), \quad x(0) = x(L) = 0\,.$$

Equivalently, the deflection $u(x)$ is the function v that minimizes the potential energy $F(v)$ in a string,

[1] `http://fenicsproject.org`

$$F(v) = \frac{1}{2}\int_0^L \left((v')^2 - fv\right)\,\mathrm{d}x\,.$$

That is, $F(u) = \min_{v\in V} F(v)$. The quantity $F(v)$ is called a functional: it takes one or more functions as input and produces a number. Loosely speaking, we may say that a functional is "a function of functions". Functionals very often involve integral expressions as above.

A range of physical problems can be formulated either as a differential equation or as a minimization of some functional. Quite often, the differential equation arises from Newton's 2nd law of motion while the functional expresses a certain kind of energy.

Many traditional applications of the finite element method, especially in solid mechanics and constructions with beams and plates, start with formulating $F(v)$ from physical principles, such as minimization of elastic energy, and then proceeds with deriving $a(u, v) = L(v)$, which is the formulation usually desired in software implementations.

The general minimization problem. The relation between a differential equation and minimization of a functional can be expressed in a general mathematical way using our abstract notation for a variational form: $a(u, v) = L(v)$. It can be shown that the variational statement

$$a(u, v) = L(v) \quad \forall v \in V,$$

is equivalent to minimizing the functional

$$F(v) = \frac{1}{2}a(v, v) - L(v)$$

over all functions $v \in V$. That is,

$$F(u) \leq F(v) \quad \forall v \in V\,.$$

Derivation. To see this, we write $F(u) \leq F(\eta)$, $\forall \eta \in V$ instead and set $\eta = u + \epsilon v$, where $v \in V$ is an arbitrary function in V. For any given arbitrary v, we can view $F(v)$ as a function $g(\epsilon)$ and find the extrema of g, which is a function of one variable. We have

$$F(\eta) = F(u + \epsilon v) = \frac{1}{2}a(u + \epsilon v, u + \epsilon v) - L(u + \epsilon v)\,.$$

From the linearity of a and L we get

$$
\begin{aligned}
g(\epsilon) &= F(u+\epsilon u) \\
&= \frac{1}{2}a(u+\epsilon v, u+\epsilon v) - L(u+\epsilon v) \\
&= \frac{1}{2}a(u, u+\epsilon v) + \frac{1}{2}\epsilon a(v, u+\epsilon u) - L(u) - \epsilon L(v) \\
&= \frac{1}{2}a(u,u) + \frac{1}{2}\epsilon a(u,v) + \frac{1}{2}\epsilon a(v,u) + \frac{1}{2}\epsilon^2 a(v,v) - L(u) - \epsilon L(v)\,.
\end{aligned}
$$

If we now assume that a is symmetric, $a(u,v) = a(v,u)$, we can write

$$
g(\epsilon) = \frac{1}{2}a(u,u) + \epsilon a(u,v) + \frac{1}{2}\epsilon^2 a(v,v) - L(u) - \epsilon L(v)\,.
$$

The extrema of g is found by searching for ϵ such that $g'(\epsilon) = 0$:

$$
g'(\epsilon) = a(u,v) - L(v) + \epsilon a(v,v) = 0\,.
$$

This linear equation in ϵ has a solution $\epsilon = (a(u,v) - L(u))/a(v,v)$ if $a(v,v) > 0$. But recall that $a(u,v) = L(v)$ for any v, so we must have $\epsilon = 0$. Since the reasoning above holds for any $v \in V$, the function $\eta = u + \epsilon v$ that makes $F(\eta)$ extreme must have $\epsilon = 0$, i.e., $\eta = u$, the solution of $a(u,v) = L(v)$ for any v in V.

Looking at $g''(\epsilon) = a(v,v)$, we realize that $\epsilon = 0$ corresponds to a unique minimum if $a(v,v) > 0$.

The equivalence of a variational form $a(u,v) = L(v)\ \forall v \in V$ and the minimization problem $F(u) \leq F(v)\ \forall v \in V$ requires that 1) a is bilinear and L is linear, 2) $a(u,v)$ is symmetric: $a(u,v) = a(v,u)$, and 3) that $a(v,v) > 0$.

Minimization of the discretized functional. Inserting $v = \sum_j c_j\psi_j$ turns minimization of $F(v)$ into minimization of a quadratic function of the parameters $c_0, \ldots, c_N$:

$$
\bar{F}(c_0, \ldots, c_N) = \sum_{j\in\mathcal{I}_s}\sum_{i\in\mathcal{I}_s} a(\psi_i, \psi_j)c_i c_j - \sum_{j\in\mathcal{I}_s} L(\psi_j)c_j
$$

of $N+1$ parameters.

Minimization of $\bar{F}$ implies

$$
\frac{\partial \bar{F}}{\partial c_i} = 0, \quad i \in \mathcal{I}_s\,.
$$

After quite some algebra one finds

$$
\sum_{j\in\mathcal{I}_s} a(\psi_i, \psi_j)c_j = L(\psi_i), \quad i \in \mathcal{I}_s,
$$

which is the same system as the one arising from $a(u,v) = L(v)$.

Calculus of variations. A branch of applied mathematics, called calculus of variations[2], deals with the technique of minimizing functionals to derive differential equations. The technique involves taking the *variation* (a kind of derivative) of functionals, which have given name to terms like variational form, variational problem, and variational formulation.

2 Examples on variational formulations

The following sections derive variational formulations for some prototype differential equations in 1D, and demonstrate how we with ease can handle variable coefficients, mixed Dirichlet and Neumann boundary conditions, first-order derivatives, and nonlinearities.

2.1 Variable coefficient

Consider the problem

$$-\frac{d}{dx}\left(\alpha(x)\frac{du}{dx}\right) = f(x), \quad x \in \Omega = [0, L],\ u(0) = C,\ u(L) = D\,. \tag{50}$$

There are two new features of this problem compared with previous examples: a variable coefficient $\alpha(x)$ and nonzero Dirichlet conditions at both boundary points.

Let us first deal with the boundary conditions. We seek

$$u(x) = B(x) + \sum_{j\in\mathcal{I}_s} c_j\psi_i(x)\,.$$

Since the Dirichlet conditions demand

$$\psi_i(0) = \psi_i(L) = 0, \quad i \in \mathcal{I}_s,$$

the function $B(x)$ must fulfill $B(0) = C$ and $B(L) = D$. The we are guaranteed that $u(0) = C$ and $u(L) = D$. How B varies in between $x = 0$ and $x = L$ is not of importance. One possible choice is

$$B(x) = C + \frac{1}{L}(D - C)x,$$

which follows from (48) with $p = 1$.

We seek $(u - B) \in V$. As usual,

$$V = \text{span}\{\psi_0, \ldots, \psi_N\}\,.$$

Note that any $v \in V$ has the property $v(0) = v(L) = 0$.

The residual arises by inserting our u in the differential equation:

[2] `https://en.wikipedia.org/wiki/Calculus_of_variations`

$$R = -\frac{d}{dx}\left(\alpha\frac{du}{dx}\right) - f\,.$$

Galerkin's method is

$$(R, v) = 0, \quad \forall v \in V,$$

or written with explicit integrals,

$$\int_\Omega \left(-\frac{d}{dx}\left(\alpha\frac{du}{dx}\right) - f\right) v\,\mathrm{d}x = 0, \quad \forall v \in V\,.$$

We proceed with integration by parts to lower the derivative from second to first order:

$$-\int_\Omega \frac{d}{dx}\left(\alpha(x)\frac{du}{dx}\right) v\,\mathrm{d}x = \int_\Omega \alpha(x)\frac{du}{dx}\frac{dv}{dx}\,\mathrm{d}x - \left[\alpha\frac{du}{dx}v\right]_0^L\,.$$

The boundary term vanishes since $v(0) = v(L) = 0$. The variational formulation is then

$$\int_\Omega \alpha(x)\frac{du}{dx}\frac{dv}{dx}\,\mathrm{d}x = \int_\Omega f(x)v\,\mathrm{d}x, \quad \forall v \in V\,.$$

The variational formulation can alternatively be written in a more compact form:

$$(\alpha u', v') = (f, v), \quad \forall v \in V\,.$$

The corresponding abstract notation reads

$$a(u, v) = L(v) \quad \forall v \in V,$$

with

$$a(u, v) = (\alpha u', v'), \quad L(v) = (f, v)\,.$$

We may insert $u = B + \sum_j c_j\psi_j$ and $v = \psi_i$ to derive the linear system:

$$(\alpha B' + \alpha\sum_{j\in\mathcal{I}_s} c_j\psi_j', \psi_i') = (f, \psi_i), \quad i \in \mathcal{I}_s\,.$$

Isolating everything with the c_j coefficients on the left-hand side and all known terms on the right-hand side gives

$$\sum_{j\in\mathcal{I}_s} (\alpha\psi_j', \psi_i')c_j = (f, \psi_i) - (\alpha(D - C)L^{-1}, \psi_i'), \quad i \in \mathcal{I}_s\,.$$

This is nothing but a linear system $\sum_j A_{i,j}c_j = b_i$ with

$$A_{i,j} = (\alpha\psi_j', \psi_i') = \int_\Omega \alpha(x)\psi_j'(x), \psi_i'(x)\,\mathrm{d}x,$$

$$b_i = (f, \psi_i) - (\alpha(D - C)L^{-1}, \psi_i') = \int_\Omega \left(f(x)\psi_i(x) - \alpha(x)\frac{D - C}{L}\psi_i'(x)\right)\,\mathrm{d}x\,.$$

2.2 First-order derivative in the equation and boundary condition

The next problem to formulate in terms of a variational form reads

$$-u''(x) + bu'(x) = f(x), \quad x \in \Omega = [0, L],\ u(0) = C,\ u'(L) = E\,. \tag{51}$$

The new features are a first-order derivative u' in the equation and the boundary condition involving the derivative: $u'(L) = E$. Since we have a Dirichlet condition at $x = 0$, we must force $\psi_i(0) = 0$ and use a boundary function to take care of the condition $u(0) = C$. Because there is no Dirichlet condition on $x = L$ we do not make any requirements to $\psi_i(L)$. The simplest possible choice of $B(x)$ is $B(x) = C$.

The expansion for u becomes

$$u = C + \sum_{j \in \mathcal{I}_s} c_j \psi_i(x)\,.$$

The variational formulation arises from multiplying the equation by a test function $v \in V$ and integrating over Ω:

$$(-u'' + bu' - f, v) = 0, \quad \forall v \in V$$

We apply integration by parts to the $u''v$ term only. Although we could also integrate $u'v$ by parts, this is not common. The result becomes

$$(u', v') + (bu', v) = (f, v) + [u'v]_0^L, \quad \forall v \in V\,.$$

Now, $v(0) = 0$ so

$$[u'v]_0^L = u'(L)v(L) = Ev(L),$$

because $u'(L) = E$. Thus, integration by parts allows us to take care of the Neumann condition in the boundary term.

Natural and essential boundary conditions.

A common mistake is to forget a boundary term like $[u'v]_0^L$ in the integration by parts. Such a mistake implies that we actually impose the condition $u' = 0$ unless there is a Dirichlet condition (i.e., $v = 0$) at that point! This fact has great practical consequences, because it is easy to forget the boundary term, and that implicitly set a boundary condition!

Since homogeneous Neumann conditions can be incorporated without "doing anything" (i.e., omitting the boundary term), and non-homogeneous Neumann conditions can just be inserted in the boundary term, such conditions are known as *natural boundary conditions*. Dirichlet conditions require more essential steps in the mathematical formulation, such as

forcing all $\varphi_i = 0$ on the boundary and constructing a $B(x)$, and are therefore known as *essential boundary conditions*.

The final variational form reads

$$(u', v') + (bu', v) = (f, v) + Ev(L), \quad \forall v \in V\,.$$

In the abstract notation we have

$$a(u, v) = L(v) \quad \forall v \in V,$$

with the particular formulas

$$a(u, v) = (u', v') + (bu', v), \quad L(v) = (f, v) + Ev(L)\,.$$

The associated linear system is derived by inserting $u = C + \sum_j c_j\psi_j$ and replacing v by ψ_i for $i \in \mathcal{I}_s$. Some algebra results in

$$\sum_{j\in\mathcal{I}_s} \underbrace{((\psi_j', \psi_i') + (b\psi_j', \psi_i))}_{A_{i,j}} c_j = \underbrace{(f, \psi_i) + E\psi_i(L)}_{b_i}\,.$$

Observe that in this problem, the coefficient matrix is not symmetric, because of the term

$$(b\psi_j', \psi_i) = \int_\Omega b\psi_j'\psi_i \,\mathrm{d}x \neq \int_\Omega b\psi_i'\psi_j \,\mathrm{d}x = (\psi_i', b\psi_j)\,.$$

2.3 Nonlinear coefficient

Finally, we show that the techniques used above to derive variational forms apply to nonlinear differential equation problems as well. Here is a model problem with a nonlinear coefficient $\alpha(u)$ and a nonlinear right-hand side $f(u)$:

$$-(\alpha(u)u')' = f(u), \quad x \in [0, L],\ u(0) = 0,\ u'(L) = E\,. \tag{52}$$

Our space V has basis $\{\psi_i\}_{i\in\mathcal{I}_s}$, and because of the condition $u(0) = 0$, we must require $\psi_i(0) = 0$, $i \in \mathcal{I}_s$.

Galerkin's method is about inserting the approximate u, multiplying the differential equation by $v \in V$, and integrate,

$$-\int_0^L \frac{d}{dx}\left(\alpha(u)\frac{du}{dx}\right) v \,\mathrm{d}x = \int_0^L f(u)v \,\mathrm{d}x \quad \forall v \in V\,.$$

The integration by parts does not differ from the case where we have $\alpha(x)$ instead of $\alpha(u)$:

$$\int_0^L \alpha(u)\frac{du}{dx}\frac{dv}{dx} \,\mathrm{d}x = \int_0^L f(u)v \,\mathrm{d}x + [\alpha(u)vu']_0^L \quad \forall v \in V\,.$$

The term $\alpha(u(0))v(0)u'(0) = 0$ since $v(0)$. The other term, $\alpha(u(L))v(L)u'(L)$, is used to impose the other boundary condition $u'(L) = E$, resulting in

$$\int_0^L \alpha(u)\frac{du}{dx}\frac{dv}{dx}\,\mathrm{d}x = \int_0^L f(u)v\,\mathrm{d}x + \alpha(u(L))v(L)E \quad \forall v \in V,$$

or alternatively written more compactly as

$$(\alpha(u)u', v') = (f(u), v) + \alpha(u(L))v(L)E \quad \forall v \in V\,.$$

Since the problem is nonlinear, we cannot identify a bilinear form $a(u, v)$ and a linear form $L(v)$. An abstract formulation is typically *find u such that*

$$F(u; v) = 0 \quad \forall v \in V,$$

with

$$F(u; v) = (a(u)u', v') - (f(u), v) - a(L)v(L)E\,.$$

By inserting $u = \sum_j c_j\psi_j$ and $v = \psi_i$ in $F(u; v)$, we get a *nonlinear system of algebraic equations* for the unknowns c_i, $i \in \mathcal{I}_s$. Such systems must be solved by constructing a sequence of linear systems whose solutions hopefully converge to the solution of the nonlinear system. Frequently applied methods are Picard iteration and Newton's method.

2.4 Computing with Dirichlet and Neumann conditions

Let us perform the necessary calculations to solve

$$-u''(x) = 2, \quad x \in \Omega = [0, 1], \quad u'(0) = C,\ u(1) = D,$$

using a global polynomial basis $\psi_i \sim x^i$. The requirement on ψ_i is that $\psi_i(1) = 0$, because u is specified at $x = 1$, so a proper set of polynomial basis functions can be

$$\psi_i(x) = (1 - x)^{i+1}, \quad i \in \mathcal{I}_s\,.$$

A suitable $B(x)$ function to handle the boundary condition $u(1) = D$ is $B(x) = Dx$. The variational formulation becomes

$$(u', v') = (2, v) - Cv(0) \quad \forall v \in V\,.$$

From inserting $u = B + \sum_j c_j\psi_j$ and choosing $v = \psi_i$ we get

$$\sum_{j\in\mathcal{I}_s}(\psi_j', \psi_i')c_j = (2, \psi_i) - (B', \psi_i') - C\psi_i(0), \quad i \in \mathcal{I}_s\,.$$

The entries in the linear system are then

$$A_{i,j} = (\psi_j', \psi_i') = \int_0^1 \psi_i'(x)\psi_j'(x)\,\mathrm{d}x = \int_0^1 (i+1)(j+1)(1-x)^{i+j}\,\mathrm{d}x = \frac{(i+1)(j+1)}{i+j+1},$$

$$\begin{aligned} b_i &= (2, \psi_i) - (D, \psi_i') - C\psi_i(0) \\ &= \int_0^1 \left(2\psi_i(x) - D\psi_i'(x)\right)\,\mathrm{d}x - C\psi_i(0) \\ &= \int_0^1 \left(2(1-x)^{i+1} - D(i+1)(1-x)^i\right)\,\mathrm{d}x - C \\ &= \frac{(D-C)(i+2)+2}{i+2} = D - C + \frac{2}{i+2}\,. \end{aligned}$$

Relevant `sympy` commands to help calculate these expressions are

```
from sympy import *
x, C, D = symbols('x C D')
i, j = symbols('i j', integer=True, positive=True)
psi_i = (1-x)**(i+1)
psi_j = psi_i.subs(i, j)
integrand = diff(psi_i, x)*diff(psi_j, x)
integrand = simplify(integrand)
A_ij = integrate(integrand, (x, 0, 1))
A_ij = simplify(A_ij)
print 'A_ij:', A_ij
f = 2
b_i = integrate(f*psi_i, (x, 0, 1)) - \
      integrate(diff(D*x, x)*diff(psi_i, x), (x, 0, 1)) - \
      C*psi_i.subs(x, 0)
b_i = simplify(b_i)
print 'b_i:', b_i
```

The output becomes

```
A_ij: (i + 1)*(j + 1)/(i + j + 1)
b_i: ((-C + D)*(i + 2) + 2)/(i + 2)
```

We can now choose some N and form the linear system, say for $N = 1$:

```
N = 1
A = zeros((N+1, N+1))
b = zeros(N+1)
print 'fresh b:', b
for r in range(N+1):
    for s in range(N+1):
        A[r,s] = A_ij.subs(i, r).subs(j, s)
    b[r,0] = b_i.subs(i, r)
```

The system becomes

$$\begin{pmatrix} 1 & 1 \\ 1 & 4/3 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 1 - C + D \\ 2/3 - C + D \end{pmatrix}$$

The solution (`c = A.LUsolve(b)`) becomes $c_0 = 2 - C + D$ and $c_1 = -1$, resulting in

$$u(x) = 1 - x^2 + D + C(x-1), \tag{53}$$

We can form this u in `sympy` and check that the differential equation and the boundary conditions are satisfied:

```
u = sum(c[r,0]*psi_i.subs(i, r) for r in range(N+1)) + D*x
print 'u:', simplify(u)
print "u'':", simplify(diff(u, x, x))
print 'BC x=0:', simplify(diff(u, x).subs(x, 0))
print 'BC x=1:', simplify(u.subs(x, 1))
```

The output becomes

```
u: C*x - C + D - x**2 + 1
u'': -2
BC x=0: C
BC x=1: D
```

The complete `sympy` code is found in `u_xx_2_CD.py`[3].

The exact solution is found by integrating twice and applying the boundary conditions, either by hand or using `sympy` as shown in Section 1.2. It appears that the numerical solution coincides with the exact one. This result is to be expected because if $(u_{\mathrm{e}} - B) \in V$, $u = u_{\mathrm{e}}$, as proved next.

2.5 When the numerical method is exact

We have some variational formulation: find $(u - B) \in V$ such that $a(u, v) = L(u)\ \forall V$. The exact solution also fulfills $a(u_{\mathrm{e}}, v) = L(v)$, but normally $(u_{\mathrm{e}} - B)$ lies in a much larger (infinite-dimensional) space. Suppose, nevertheless, that $u_{\mathrm{e}} - B = E$, where $E \in V$. That is, apart from Dirichlet conditions, u_{e} lies in our finite-dimensional space V we use to compute u. Writing also u on the same form $u = B + F$, $F \in V$, we have

$$\begin{aligned} a(B + E, v) &= L(v) \quad \forall v \in V, \\ a(B + F, v) &= L(v) \quad \forall v \in V\,. \end{aligned}$$

Since these are two variational statements in the same space, we can subtract them and use the bilinear property of $a(\cdot, \cdot)$:

$$\begin{aligned} a(B + E, v) - a(B + F, v) &= L(v) - L(v) \\ a(B + E - (B + F), v) &= 0 \\ a(E - F), v) &= 0 \end{aligned}$$

If $a(E - F), v) = 0$ for all v in V, then $E - F$ must be zero everywhere in the domain, i.e., $E = F$. Or in other words: $u = u_{\mathrm{e}}$. This proves that the exact

[3] `http://tinyurl.com/nm5587k/varform/u_xx_2_CD.py`

solution is recovered if $u_{\mathrm{e}} - B$ lies in V., i.e., can expressed as $\sum_{j\in\mathcal{I}_s} d_j\psi_j$ if $\{\psi_j\}_{j\in\mathcal{I}_s}$ is a basis for V. The method will then compute the solution $c_j = d_j$, $j \in \mathcal{I}_s$.

The case treated in Section 2.4 is of the type where $u_{\mathrm{e}} - B$ is a quadratic function that is 0 at $x = 1$, and therefore $(u_{\mathrm{e}} - B) \in V$, and the method finds the exact solution.

3 Computing with finite elements

The purpose of this section is to demonstrate in detail how the finite element method can the be applied to the model problem

$$-u''(x) = 2, \quad x \in (0, L),\ u(0) = u(L) = 0,$$

with variational formulation

$$(u', v') = (2, v) \quad \forall v \in V\,.$$

Any $v \in V$ must obey $v(0) = v(L) = 0$ because of the Dirichlet conditions on u. The variational formulation is derived in Section 1.10.

3.1 Finite element mesh and basis functions

We introduce a finite element mesh with N_e cells, all with length h, and number the cells from left to right. Choosing P1 elements, there are two nodes per cell, and the coordinates of the nodes become

$$x_i = ih, \quad h = L/N_e, \quad i = 0, \ldots, N_n - 1 = N_e\,.$$

Any node i is associated with a finite element basis function $\varphi_i(x)$. When approximating a given function f by a finite element function u, we expand u using finite element basis functions associated with *all* nodes in the mesh. The parameter N, which counts the unknowns from 0 to N, is then equal to $N_n - 1$ such that the total number of unknowns, $N + 1$, is the total number of nodes. However, when solving differential equations we will often have $N < N_n - 1$ because of Dirichlet boundary conditions. The reason is simple: we know what u are at some (here two) nodes, and the number of unknown parameters is naturally reduced.

In our case with homogeneous Dirichlet boundary conditions, we do not need any boundary function $B(x)$, so we can work with the expansion

$$u(x) = \sum_{j\in\mathcal{I}_s} c_j\psi_j(x)\,. \tag{54}$$

Because of the boundary conditions, we must demand $\psi_i(0) = \psi_i(L) = 0$, $i \in \mathcal{I}_s$. When ψ_i for all $i = 0, \ldots, N$ is to be selected among the finite element basis functions φ_j, $j = 0, \ldots, N_n - 1$, we have to avoid using φ_j functions that do not

vanish at $x_0 = 0$ and $x_{N_n-1} = L$. However, all φ_j vanish at these two nodes for $j = 1, \ldots, N_n - 2$. Only basis functions associated with the end nodes, φ_0 and φ_{N_n-1}, violate the boundary conditions of our differential equation. Therefore, we select the basis functions φ_i to be the set of finite element basis functions associated with all the interior nodes in the mesh:

$$\psi_i = \varphi_{i+1}, \quad i = 0, \ldots, N\,.$$

The i index runs over all the unknowns c_i in the expansion for u, and in this case $N = N_n - 3$.

In the general case, the nodes are not necessarily numbered from left to right, so we introduce a mapping from the node numbering, or more precisely the degree of freedom numbering, to the numbering of the unknowns in the final equation system. These unknowns take on the numbers $0, \ldots, N$. Unknown number j in the linear system corresponds to degree of freedom number $\nu(j)$, $j \in \mathcal{I}_s$. We can then write

$$\psi_i = \varphi_{\nu(i)}, \quad i = 0, \ldots, N\,.$$

With a regular numbering as in the present example, $\nu(j) = j+1$, $j = 0, \ldots, N = N_n - 3$.

3.2 Computation in the global physical domain

We shall first perform a computation in the x coordinate system because the integrals can be easily computed here by simple, visual, geometric considerations. This is called a global approach since we work in the x coordinate system and compute integrals on the global domain $[0, L]$.

The entries in the coefficient matrix and right-hand side are

$$A_{i,j} = \int_0^L \psi_i'(x)\psi_j'(x)\,\mathrm{d}x, \quad b_i = \int_0^L 2\psi_i(x)\,\mathrm{d}x, \quad i,j \in \mathcal{I}_s\,.$$

Expressed in terms of finite element basis functions φ_i we get the alternative expressions

$$A_{i,j} = \int_0^L \varphi_{i+1}'(x)\varphi_{j+1}'(x)\,\mathrm{d}x, \quad b_i = \int_0^L 2\varphi_{i+1}(x)\,\mathrm{d}x, \quad i,j \in \mathcal{I}_s\,.$$

For the following calculations the subscripts on the finite element basis functions are more conveniently written as i and j instead of $i + 1$ and $j + 1$, so our notation becomes

$$A_{i-1,j-1} = \int_0^L \varphi_i'(x)\varphi_j'(x)\,\mathrm{d}x, \quad b_{i-1} = \int_0^L 2\varphi_i(x)\,\mathrm{d}x,$$

where the i and j indices run as $i, j = 1, \ldots, N + 1 = N_n - 2$.

The $\varphi_i(x)$ function is a hat function with peak at $x = x_i$ and a linear variation in $[x_{i-1}, x_i]$ and $[x_i, x_{i+1}]$. The derivative is $1/h$ to the left of x_i and $-1/h$ to the right, or more formally,

$$\varphi_i'(x) = \begin{cases} 0, & x < x_{i-1}, \\ h^{-1}, & x_{i-1} \leq x < x_i, \\ -h^{-1}, & x_i \leq x < x_{i+1}, \\ 0, & x \geq x_{i+1} \end{cases} \tag{55}$$

Figure 2 shows $\varphi_2'(x)$ and $\varphi_3'(x)$.

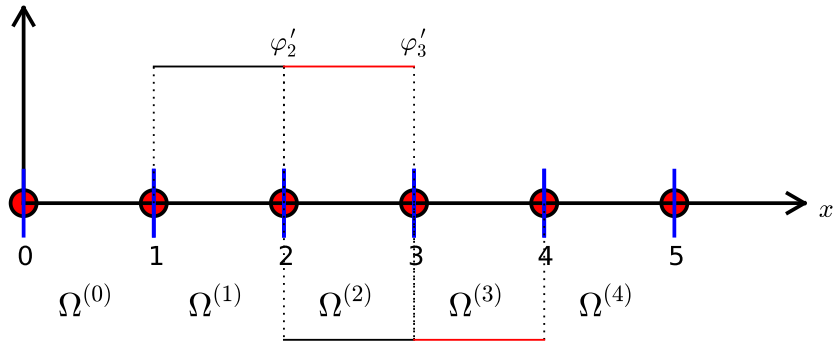


Figure 2: Illustration of the derivative of piecewise linear basis functions associated with nodes in cell 2.

We realize that φ_i' and φ_j' have no overlap, and hence their product vanishes, unless i and j are nodes belonging to the same cell. The only nonzero contributions to the coefficient matrix are therefore

$$\begin{aligned} A_{i-1,i-2} &= \int_0^L \varphi_i'(x)\varphi_{i-1}'(x)\,\mathrm{d}x, \\ A_{i-1,i-1} &= \int_0^L \varphi_i'(x)^2\,\mathrm{d}x, \\ A_{i-1,i} &= \int_0^L \varphi_i'(x)\varphi_{i+1}'(x)\,\mathrm{d}x, \end{aligned}$$

for $i = 1, \ldots, N+1$, but for $i = 1$, $A_{i-1,i-2}$ is not defined, and for $i = N+1$, $A_{i-1,i}$ is not defined.

From Figure 2, we see that $\varphi_{i-1}'(x)$ and $\varphi_i'(x)$ have overlap of one cell $\Omega^{(i-1)} = [x_{i-1}, x_i]$ and that their product then is $-1/h^2$. The integrand is constant and therefore $A_{i-1,i-2} = -h^{-2}h = -h^{-1}$. A similar reasoning can be applied to $A_{i-1,i}$, which also becomes $-h^{-1}$. The integral of $\varphi_i'(x)^2$ gets contributions from two cells, $\Omega^{(i-1)} = [x_{i-1}, x_i]$ and $\Omega^{(i)} = [x_i, x_{i+1}]$, but $\varphi_i'(x)^2 = h^{-2}$ in both cells, and the length of the integration interval is $2h$ so we get $A_{i-1,i-1} = 2h^{-1}$.

The right-hand side involves an integral of $2\varphi_i(x)$, $i = 1, \ldots, N_n - 2$, which is just the area under a hat function of height 1 and width $2h$, i.e., equal to h. Hence, $b_{i-1} = 2h$.

To summarize the linear system, we switch from i to $i+1$ such that we can write

$$A_{i,i-1} = A_{i,i+1} = -h^{-1}, \quad A_{i,i} = 2h^{-1}, \quad b_i = 2h\,.$$

The equation system to be solved only involves the unknowns c_i for $i \in \mathcal{I}_s$. With our numbering of unknowns and nodes, we have that c_i equals $u(x_{i+1})$. The complete matrix system then takes the following form:

$$\frac{1}{h}\begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \end{pmatrix} \tag{56}$$

3.3 Comparison with a finite difference discretization

A typical row in the matrix system (56) can be written as

$$-\frac{1}{h}c_{i-1} + \frac{2}{h}c_i - \frac{1}{h}c_{i+1} = 2h\,. \tag{57}$$

Let us introduce the notation u_j for the value of u at node j: $u_j = u(x_j)$, since we have the interpretation $u(x_j) = \sum_j c_j\varphi(x_j) = \sum_j c_j\delta_{ij} = c_j$. The unknowns $c_0, \ldots, c_N$ are $u_1, \ldots, u_{N_n-2}$. Shifting i with $i+1$ in (57) and inserting $u_i = c_{i-1}$, we get

$$-\frac{1}{h}u_{i-1} + \frac{2}{h}u_i - \frac{1}{h}u_{i+1} = 2h, \tag{58}$$

A finite difference discretization of $-u''(x) = 2$ by a centered, second-order finite difference approximation $u''(x_i) \approx [D_xD_xu]_i$ with $\Delta x = h$ yields

$$-\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = 2, \tag{59}$$

which is, in fact, equivalent to (58) if (58) is divided by h. Therefore, the finite difference and the finite element method are equivalent in this simple test problem.

Sometimes a finite element method generates the finite difference equations on a uniform mesh, and sometimes the finite element method generates equations that are different. The differences are modest, but may influence the numerical quality of the solution significantly, especially in time-dependent problems. It depends on the problem at hand whether a finite element discretization is more or less accurate than a corresponding finite difference discretization.

3.4 Cellwise computations

Software for finite element computations normally employs the cell by cell computational procedure where an element matrix and vector are calculated for each cell and assembled in the global linear system. Let us go through the details of this type of algorithm.

All integrals are mapped to the local reference coordinate system $X \in [-1, 1]$. In the present case, the matrix entries contain derivatives with respect to x,

$$A^{(e)}_{i-1,j-1} = \int_{\Omega^{(e)}} \varphi_i'(x)\varphi_j'(x)\,\mathrm{d}x = \int_{-1}^{1} \frac{d}{dx}\tilde{\varphi}_r(X)\frac{d}{dx}\tilde{\varphi}_s(X)\frac{h}{2}\,\mathrm{d}X,$$

where the global degree of freedom i is related to the local degree of freedom r through $i = q(e, r)$. Similarly, $j = q(e, s)$. The local degrees of freedom run as $r, s = 0, 1$ for a P1 element.

The integral for the element matrix. There are simple formulas for the basis functions $\tilde{\varphi}_r(X)$ as functions of X. However, we now need to find the derivative of $\tilde{\varphi}_r(X)$ with respect to x. Given

$$\tilde{\varphi}_0(X) = \frac{1}{2}(1 - X), \quad \tilde{\varphi}_1(X) = \frac{1}{2}(1 + X),$$

we can easily compute $d\tilde{\varphi}_r/dX$:

$$\frac{d\tilde{\varphi}_0}{dX} = -\frac{1}{2}, \quad \frac{d\tilde{\varphi}_1}{dX} = \frac{1}{2}\,.$$

From the chain rule,

$$\frac{d\tilde{\varphi}_r}{dx} = \frac{d\tilde{\varphi}_r}{dX}\frac{dX}{dx} = \frac{2}{h}\frac{d\tilde{\varphi}_r}{dX}\,. \tag{60}$$

The transformed integral is then

$$A^{(e)}_{i-1,j-1} = \int_{\Omega^{(e)}} \varphi_i'(x)\varphi_j'(x)\,\mathrm{d}x = \int_{-1}^{1} \frac{2}{h}\frac{d\tilde{\varphi}_r}{dX}\frac{2}{h}\frac{d\tilde{\varphi}_s}{dX}\frac{h}{2}\,\mathrm{d}X\,.$$

The integral for the element vector. The right-hand side is transformed according to

$$b_{i-1}^{(e)} = \int_{\Omega^{(e)}} 2\varphi_i(x)\,\mathrm{d}x = \int_{-1}^{1} 2\tilde{\varphi}_r(X)\frac{h}{2}\,\mathrm{d}X, \quad i = q(e,r),\ r = 0,1\,.$$

Detailed calculations of the element matrix and vector. Specifically for P1 elements we arrive at the following calculations for the element matrix entries:

$$\begin{aligned}
\tilde{A}_{0,0}^{(e)} &= \int_{-1}^{1} \frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}\left(-\frac{1}{2}\right)\frac{h}{2}\,\mathrm{d}X = \frac{1}{h} \\
\tilde{A}_{0,1}^{(e)} &= \int_{-1}^{1} \frac{2}{h}\left(-\frac{1}{2}\right)\frac{2}{h}\left(\frac{1}{2}\right)\frac{h}{2}\,\mathrm{d}X = -\frac{1}{h} \\
\tilde{A}_{1,0}^{(e)} &= \int_{-1}^{1} \frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}\left(-\frac{1}{2}\right)\frac{h}{2}\,\mathrm{d}X = -\frac{1}{h} \\
\tilde{A}_{1,1}^{(e)} &= \int_{-1}^{1} \frac{2}{h}\left(\frac{1}{2}\right)\frac{2}{h}\left(\frac{1}{2}\right)\frac{h}{2}\,\mathrm{d}X = \frac{1}{h}
\end{aligned}$$

The element vector entries become

$$\begin{aligned}
\tilde{b}_0^{(e)} &= \int_{-1}^{1} 2\frac{1}{2}(1-X)\frac{h}{2}\,\mathrm{d}X = h \\
\tilde{b}_1^{(e)} &= \int_{-1}^{1} 2\frac{1}{2}(1+X)\frac{h}{2}\,\mathrm{d}X = h\,.
\end{aligned}$$

Expressing these entries in matrix and vector notation, we have

$$\tilde{A}^{(e)} = \frac{1}{h}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h\begin{pmatrix} 1 \\ 1 \end{pmatrix}. \tag{61}$$

Contributions from the first and last cell. The first and last cell involve only one unknown and one basis function because of the Dirichlet boundary conditions at the first and last node. The element matrix therefore becomes a 1×1 matrix and there is only one entry in the element vector. On cell 0, only $\psi_0 = \varphi_1$ is involved, corresponding to integration with $\tilde{\varphi}_1$. On cell N_e, only $\psi_N = \varphi_{N_n-2}$ is involved, corresponding to integration with $\tilde{\varphi}_0$. We then get the special end-cell contributions

$$\tilde{A}^{(e)} = \frac{1}{h}\begin{pmatrix} 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h\begin{pmatrix} 1 \end{pmatrix}, \tag{62}$$

for $e = 0$ and $e = N_e$. In these cells, we have only one degree of freedom, not two as in the interior cells.

Assembly. The next step is to assemble the contributions from the various cells. The assembly of an element matrix and vector into the global matrix and right-hand side can be expressed as

$$A_{q(e,r),q(e,s)} = A_{q(e,r),q(e,s)} + \tilde{A}^{(e)}_{r,s}, \quad b_{q(e,r)} = b_{q(e,r)} + \tilde{b}^{(e)}_r,$$

for r and s running over all local degrees of freedom in cell e.

To make the assembly algorithm more precise, it is convenient to set up Python data structures and a code snippet for carrying out all details of the algorithm. For a mesh of four equal-sized P1 elements and $L = 2$ we have

```
vertices = [0, 0.5, 1, 1.5, 2]
cells = [[0, 1], [1, 2], [2, 3], [3, 4]]
dof_map = [[0], [0, 1], [1, 2], [2]]
```

The total number of degrees of freedom is 3, being the function values at the internal 3 nodes where u is unknown. In cell 0 we have global degree of freedom 0, the next cell has u unknown at its two nodes, which become global degrees of freedom 0 and 1, and so forth according to the `dof_map` list. The mathematical $q(e, r)$ quantity is nothing but the `dof_map` list.

Assume all element matrices are stored in a list `Ae` such that `Ae[e][i,j]` is $\tilde{A}^{(e)}_{i,j}$. A corresponding list for the element vectors is named `be`, where `be[e][r]` is $\tilde{b}^{(e)}_r$. A Python code snippet illustrates all details of the assembly algorithm:

```
# A[i,j]: coefficient matrix, b[i]: right-hand side
for e in range(len(Ae)):
    for r in range(Ae[e].shape[0]):
        for s in range(Ae[e].shape[1]):
            A[dof_map[e,r],dof_map[e,s]] += Ae[e][i,j]
        b[dof_map[e,r]] += be[e][i,j]
```

The general case with `N_e` P1 elements of length `h` has

```
N_n = N_e + 1
vertices = [i*h for i in range(N_n)]
cells = [[e, e+1] for e in range(N_e)]
dof_map = [[0]] + [[e-1, e] for i in range(1, N_e)] + [[N_n-2]]
```

Carrying out the assembly results in a linear system that is identical to (56), which is not surprising, since the procedures is mathematically equivalent to the calculations in the physical domain.

So far, our technique for computing the matrix system have assumed that $u(0) = u(L) = 0$. The next section deals with the extension to nonzero Dirichlet conditions.

4 Boundary conditions: specified nonzero value

We have to take special actions to incorporate nonzero Dirichlet conditions, such as $u(L) = D$, into the computational procedures. The present section outlines alternative, yet mathematically equivalent, methods.

4.1 General construction of a boundary function

In Section 1.11 we introduced a boundary function $B(x)$ to deal with nonzero Dirichlet boundary conditions for u. The construction of such a function is not always trivial, especially not in multiple dimensions. However, a simple and general construction idea exists when the basis functions have the property

$$\varphi_i(x_j) = \delta_{ij}, \quad \delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & i \neq j, \end{cases}$$

where x_j is a boundary point. Examples on such functions are the Lagrange interpolating polynomials and finite element functions.

Suppose now that u has Dirichlet boundary conditions at nodes with numbers $i \in I_b$. For example, $I_b = \{0, N_n - 1\}$ in a 1D mesh with node numbering from left to right and Dirichlet conditions at the end nodes $i = 0$ and $i = N_n - 1$. Let U_i be the corresponding prescribed values of $u(x_i)$. We can then, in general, use

$$B(x) = \sum_{j \in I_b} U_j \varphi_j(x)\,. \tag{63}$$

It is easy to verify that $B(x_i) = \sum_{j \in I_b} U_j \varphi_j(x_i) = U_i$.

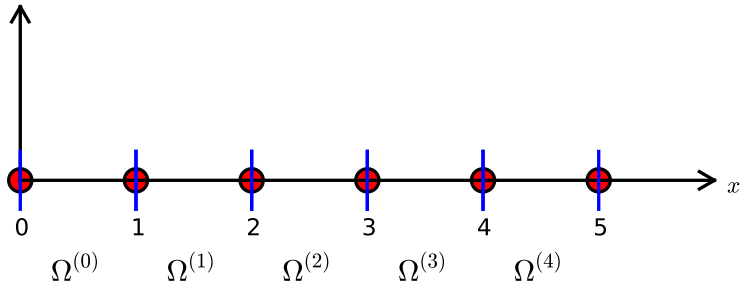
The unknown function can then be written as

$$u(x) = \sum_{j \in I_b} U_j \varphi_j(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)}, \tag{64}$$

where $\nu(j)$ maps unknown number j in the equation system to node $\nu(j)$, I_b is the set of indices corresponding to basis functions associated with nodes where Dirichlet conditions apply, and $\mathcal{I}_s$ is the set of indices used to number the unknowns from zero to N. We can easily show that with this u, a Dirichlet condition $u(x_k) = U_k$ is fulfilled:

$$u(x_k) = \sum_{j \in I_b} U_j \underbrace{\varphi_j(x_k)}_{\neq 0 \Leftrightarrow j=k} + \sum_{j \in \mathcal{I}_s} c_j \underbrace{\varphi_{\nu(j)}(x_k)}_{=0,\ k \notin \mathcal{I}_s} = U_k$$

Some examples will further clarify the notation. With a regular left-to-right numbering of nodes in a mesh with P1 elements, and Dirichlet conditions at $x = 0$, we use finite element basis functions associated with the nodes $1, 2, \ldots, N_n - 1$, implying that $\nu(j) = j + 1$, $j = 0, \ldots, N$, where $N = N_n - 2$. Consider a particular mesh:



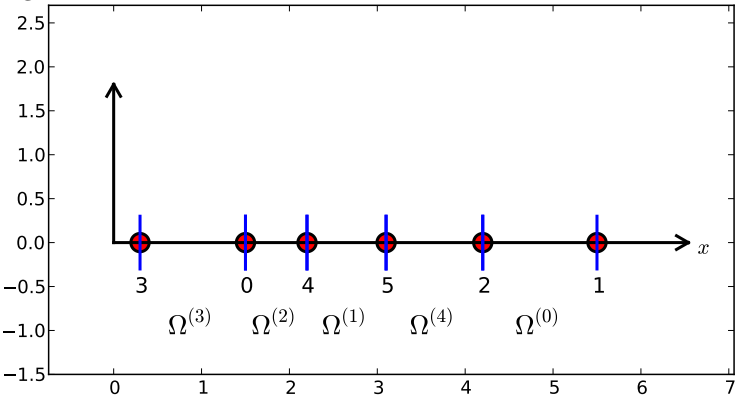
The expansion associated with this mesh becomes

$$u(x) = U_0\varphi_0(x) + c_0\varphi_1(x) + c_1\varphi_2(x) + \cdots + c_4\varphi_5(x)\,.$$

Switching to the more standard case of left-to-right numbering and boundary conditions $u(0) = C$, $u(L) = D$, we have $N = N_n - 3$ and

$$\begin{aligned} u(x) &= C\varphi_0 + D\varphi_{N_n-1} + \sum_{j\in\mathcal{I}_s} c_j\varphi_{j+1} \\ &= C\varphi_0 + D\varphi_{N_n} + c_0\varphi_1 + c_1\varphi_2 + \cdots + c_N\varphi_{N_n-2}\,. \end{aligned}$$

Finite element meshes in non-trivial 2D and 3D geometries usually lead to an irregular cell and node numbering. Let us therefore take a look at an irregular numbering in 1D:



Say we in this mesh have Dirichlet conditions on the left-most and right-most node, with numbers 3 and 1, respectively. We can number the unknowns at the interior nodes as we want, e.g., from left to right, resulting in $\nu(0) = 0$, $\nu(1) = 4$, $\nu(2) = 5$, $\nu(3) = 2$. This gives

$$B(x) = U_3\varphi_3(x) + U_1\varphi_1(x),$$

and

$$u(x) = B(x) + \sum_{j=0}^{3} c_j\varphi_{\nu(j)} = U_3\varphi_3 + U_1\varphi_1 + c_0\varphi_0 + c_1\varphi_4 + c_2\varphi_5 + c_3\varphi_2\,.$$

The idea of constructing B, described here, generalizes almost trivially to 2D and 3D problems: $B = \sum_{j\in I_b} U_j\varphi_j$, where I_b is the index set containing the numbers of all the nodes on the boundaries where Dirichlet values are prescribed.

4.2 Example on computing with a finite element-based boundary function

Let us see how the model problem $-u'' = 2$, $u(0) = C$, $u(L) = D$, is affected by a $B(x)$ to incorporate boundary values. Inserting the expression

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)$$

in $-(u'', \psi_i) = (f, \psi_i)$ and integrating by parts results in a linear system with

$$A_{i,j} = \int_0^L \psi_i'(x)\psi_j'(x)\,\mathrm{d}x, \quad b_i = \int_0^L (f(x)\psi_i(x) - B'(x)\psi_i'(x))\,\mathrm{d}x\,.$$

We choose $\psi_i = \varphi_{i+1}$, $i = 0, \ldots, N = N_n - 3$ if the node numbering is from left to right. (Later we also need the assumption that cells too are numbered from left to right.) The boundary function becomes

$$B(x) = C\varphi_0(x) + D\varphi_{N_n-1}(x)\,.$$

The expansion for $u(x)$ is

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_{j+1}(x)\,.$$

We can write the matrix and right-hand side entries as

$$\begin{aligned} A_{i-1,j-1} &= \int_0^L \varphi_i'(x)\varphi_j'(x)\,\mathrm{d}x, \\ b_{i-1} &= \int_0^L (f(x)\varphi_i(x) - (C\varphi_0'(x) + D\varphi_{N_n-1}'(x))\varphi_i'(x))\,\mathrm{d}x, \end{aligned}$$

for $i, j = 1, \ldots, N+1 = N_n - 2$. Note that we have here used $B' = C\varphi_0' + D\varphi_{N_n-1}'$.

Computations in physical coordinates. Most of the terms in the linear system have already been computed so we concentrate on the new contribution from the boundary function. The integral $C\int_0^L \varphi_0'(x))\varphi_i'(x)\,\mathrm{d}x$ can only get a nonzero contribution from the first cell, $\Omega^{(0)} = [x_0, x_1]$ since $\varphi_0'(x) = 0$ on all other cells. Moreover, $\varphi_0'(x)\varphi_i'(x)\,\mathrm{d}x \neq 0$ only for $i = 0$ and $i = 1$ (but node $i = 0$ is excluded from the formulation), since $\varphi_i = 0$ on the first cell if $i > 1$. With a similar reasoning we realize that $D\int_0^L \varphi_{N_n-1}'(x))\varphi_i'(x)\,\mathrm{d}x$ can only get a nonzero contribution from the last cell. From the explanations of the calculations in Section 3.6 in [5] we then find that

$$\int_0^L \varphi_0'(x)\varphi_1'(x)\,\mathrm{d}x = (-\frac{1}{h})\cdot\frac{1}{h}\cdot h = -\frac{1}{h},$$
$$\int_0^L \varphi_{N_n-1}'(x)\varphi_{N_n-2}'(x)\,\mathrm{d}x = \frac{1}{h}\cdot(-\frac{1}{h})\cdot h = -\frac{1}{h}\,.$$

With these expressions we get

$$b_0 = \int_0^L f(x)\varphi_1\,\mathrm{d}x - C(-\frac{1}{h}), \quad b_N = \int_0^L f(x)\varphi_{N_n-2}\,\mathrm{d}x - D(-\frac{1}{h})\,.$$

Cellwise computations on the reference element. As an equivalent alternative, we now turn to cellwise computations. The element matrices and vectors are calculated as in Section 3.4, so we concentrate on the impact of the new term involving $B(x)$. This new term, $B' = C\varphi_0' + D\varphi_{N_n-1}'$, vanishes on all cells except for $e = 0$ and $e = N_e$. Over the first cell ($e = 0$) the $B'(x)$ function in local coordinates reads

$$\frac{dB}{dx} = C\frac{2}{h}\frac{d\tilde{\varphi}_0}{dX},$$

while over the last cell ($e = N_e$) it looks like

$$\frac{dB}{dx} = D\frac{2}{h}\frac{d\tilde{\varphi}_1}{dX}\,.$$

For an arbitrary interior cell, we have the formula

$$\tilde{b}_r^{(e)} = \int_{-1}^1 f(x(X))\tilde{\varphi}_r(X)\frac{h}{2}\,\mathrm{d}X,$$

for an entry in the local element vector. In the first cell, the value at local node 0 is known so only the value at local node 1 is unknown. The associated element vector entry becomes

$$\tilde{b}_0^{(1)} = \int_{-1}^1 \left(f\tilde{\varphi}_1 - C\frac{2}{h}\frac{d\tilde{\varphi}_0}{dX}\frac{2}{h}\frac{d\tilde{\varphi}_1}{dX}\right)\frac{h}{2}\,\mathrm{d}X = \frac{h}{2}2\int_{-1}^1 \tilde{\varphi}_1\,\mathrm{d}X - C\frac{2}{h}(-\frac{1}{2})\frac{2}{h}\frac{1}{2}\frac{h}{2}\cdot 2 = h + C\frac{1}{h}\,.$$

The value at local node 1 in the last cell is known so the element vector here is

$$\tilde{b}_0^{N_e} = \int_{-1}^1 \left(f\tilde{\varphi}_0 - D\frac{2}{h}\frac{d\tilde{\varphi}_1}{dX}\frac{2}{h}\frac{d\tilde{\varphi}_0}{dX}\right)\frac{h}{2}\,\mathrm{d}X = \frac{h}{2}2\int_{-1}^1 \tilde{\varphi}_0\,\mathrm{d}X - D\frac{2}{h}\frac{1}{2}\frac{2}{h}(-\frac{1}{2})\frac{h}{2}\cdot 2 = h + D\frac{1}{h}\,.$$

The contributions from the $B(x)$ function to the global right-hand side vector becomes C/h for b_0 and D/h for b_N, exactly as we computed in the physical domain.

4.3 Modification of the linear system

From an implementational point of view, there is a convenient alternative to adding the $B(x)$ function and using only the basis functions associated with nodes where u is truly unknown. Instead of seeking

$$u(x) = \sum_{j \in I_b} U_j \varphi_j(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\nu(j)}(x), \tag{65}$$

we use the sum over all degrees of freedom, including the known boundary values:

$$u(x) = \sum_{j \in \mathcal{I}_s} c_j \varphi_j(x)\,. \tag{66}$$

Note that the collections of unknowns $\{c_i\}_{i \in \mathcal{I}_s}$ in (65) and (66) are different. The index set $\mathcal{I}_s = \{0, \ldots, N\}$ always goes to N, and the number of unknowns is $N+1$, but in (65) the unknowns correspond to nodes where u is not known, while in (66) the unknowns cover u values at all the nodes. So, if the index set I_b contains N_b node numbers where u is prescribed, we have that $N = N_n - N_b$ in (65) and $N = N_n$ in (66).

The idea is to compute the entries in the linear system as if no Dirichlet values are prescribed. Afterwards, we modify the linear system to ensure that the known c_j values are incorporated.

A potential problem arises for the boundary term $[u'v]_0^L$ from the integration by parts: imagining no Dirichlet conditions means that we no longer require $v = 0$ at Dirichlet points, and the boundary term is then nonzero at these points. However, when we modify the linear system, we will erase whatever the contribution from $[u'v]_0^L$ should be at the Dirichlet points in the right-hand side of the linear system. We can therefore safely forget $[u'v]_0^L$ at any point where a Dirichlet condition applies.

Computations in the physical system. Let us redo the computations in the example in Section 4.1. We solve $-u'' = 2$ with $u(0) = 0$ and $u(L) = D$. The expressions for $A_{i,j}$ and b_i are the same, but the numbering is different as the numbering of unknowns and nodes now coincide:

$$A_{i,j} = \int_0^L \varphi_i'(x)\varphi_j'(x)\,\mathrm{d}x, \quad b_i = \int_0^L f(x)\varphi_i(x)\,\mathrm{d}x,$$

for $i, j = 0, \ldots, N = N_n - 1$. The integrals involving basis functions corresponding to interior mesh nodes, $i, j = 1, \ldots, N_n - 2$, are obviously the same as before. We concentrate on the contributions from φ_0 and φ_{N_n-1}:

$$
\begin{aligned}
A_{0,0} &= \int_0^L (\varphi_0')^2 \,\mathrm{d}x = \int_0^{x_1} = (\varphi_0')^2 \,\mathrm{d}x \frac{1}{h}, \\
A_{0,1} &= \int_0^L \varphi_0' \varphi_1' \,\mathrm{d}x = \int_0^{x_1} \varphi_0' \varphi_1' \,\mathrm{d}x = -\frac{1}{h}, \\
A_{N,N} &= \int_0^L (\varphi_N')^2 \,\mathrm{d}x = \int_{x_{N_n-2}}^{x_{N_n-1}} (\varphi_N')^2 \,\mathrm{d}x = \frac{1}{h}, \\
A_{N,N-1} &= \int_0^L \varphi_N' \varphi_{N-1}' \,\mathrm{d}x = \int_{x_{N_n-2}}^{x_{N_n-1}} \varphi_N' \varphi_{N-1}' \,\mathrm{d}x = -\frac{1}{h}\,.
\end{aligned}
$$

The new terms on the right-hand side are also those involving φ_0 and φ_{N_n-1}:

$$
\begin{aligned}
b_0 &= \int_0^L 2\varphi_0(x) \,\mathrm{d}x = \int_0^{x_1} 2\varphi_0(x) \,\mathrm{d}x = h, \\
b_N &= \int_0^L 2\varphi_{N_n-1} \,\mathrm{d}x = \int_{x_{N_n-2}}^{x_{N_n-1}} 2\varphi_{N_n-1} \,\mathrm{d}x = h\,.
\end{aligned}
$$

The complete matrix system, involving all degrees of freedom, takes the form

$$
\frac{1}{h}
\begin{pmatrix}
1 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
-1 & 2 & -1 & \ddots & & & & & \vdots \\
0 & -1 & 2 & -1 & \ddots & & & & \vdots \\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\
\vdots & & & & & \ddots & \ddots & \ddots & -1 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 1
\end{pmatrix}
\begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix}
=
\begin{pmatrix} h \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ h \end{pmatrix}
\tag{67}
$$

Incorporation of Dirichlet values can now be done by replacing the first and last equation by the very simple equations $c_0 = 0$ and $c_N = D$, respectively. Note that the factor $1/h$ in front of the matrix then requires a factor h to be introduce appropriately on the diagonal in the first and last row of the matrix.

$$\frac{1}{h}\begin{pmatrix} h & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 & h \end{pmatrix}\begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 0 \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ D \end{pmatrix} \tag{68}$$

Note that because we do not require $\varphi_i(0) = 0$ and $\varphi_i(L) = 0$, $i \in \mathcal{I}_s$, the boundary term $[u'v]_0^L$, in principle, gives contributions $u'(0)\varphi_0(0)$ to b_0 and $u'(L)\varphi_N(L)$ to b_N ($u'\varphi_i$ vanishes for $x = 0$ or $x = L$ for $i = 1, \ldots, N-1$). Nevertheless, we erase these contributions in b_0 and b_N and insert boundary values instead. This argument shows why we can drop computing $[u'v]_0^L$ at Dirichlet nodes when we implement the Dirichlet values by modifying the linear system.

4.4 Symmetric modification of the linear system

The original matrix system (56) is symmetric, but the modifications in (68) destroy this symmetry. Our described modification will in general destroy an initial symmetry in the matrix system. This is not a particular computational disadvantage for tridiagonal systems arising in 1D problems, but may be more serious in 2D and 3D problems when the systems are large and exploiting symmetry can be important for halving the storage demands and speeding up computations. Methods for solving symmetric matrix are also usually more stable and efficient than those for non-symmetric systems. Therefore, an alternative modification which preserves symmetry is attractive.

One can formulate a general algorithm for incorporating a Dirichlet condition in a symmetric way. Let c_k be a coefficient corresponding to a known value $u(x_k) = U_k$. We want to replace equation k in the system by $c_k = U_k$, i.e., insert zeroes in row number k in the coefficient matrix, set 1 on the diagonal, and replace b_k by U_k. A symmetry-preserving modification consists in first subtracting column number k in the coefficient matrix, i.e., $A_{i,k}$ for $i \in \mathcal{I}_s$, times the boundary value U_k, from the right-hand side: $b_i \leftarrow b_i - A_{i,k}U_k$, $i = 0, \ldots, N$. Then we put zeroes in row number k *and* column number k in the coefficient matrix, and finally set $b_k = U_k$. The steps in algorithmic form becomes

1. $b_i \leftarrow b_i - A_{i,k}U_k$ for $i \in \mathcal{I}_s$

2. $A_{i,k} = A_{k,i} = 0$ for $i \in \mathcal{I}_s$

3. $A_{k,k} = 1$

4. $b_i = U_k$

This modification goes as follows for the specific linear system written out in (67) in Section 4.3. First we subtract the first column in the coefficient matrix, times the boundary value, from the right-hand side. Because $c_0 = 0$, this subtraction has no effect. Then we subtract the last column, times the boundary value D, from the right-hand side. This action results in $b_{N-1} = 2h + D/h$ and $b_N = h - 2D/h$. Thereafter, we place zeros in the first and last row and column in the coefficient matrix and 1 on the two corresponding diagonal entries. Finally, we set $b_0 = 0$ and $b_N = D$. The result becomes

$$\frac{1}{h}\begin{pmatrix} h & 0 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 & h \end{pmatrix}\begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} 0 \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h + D/h \\ D \end{pmatrix} \tag{69}$$

4.5 Modification of the element matrix and vector

The modifications of the global linear system can alternatively be done for the element matrix and vector. Let us perform the associated calculations in the computational example where the element matrix and vector is given by (61). The modifications are needed in cells where one of the degrees of freedom is known. In the present example, this means the first and last cell. We compute the element matrix and vector as if there were no Dirichlet conditions. The boundary term $[u'v]_0^L$ is simply forgotten at nodes that have Dirichlet conditions because the modification of the element vector will anyway erase the contribution from the boundary term. In the first cell, local degree of freedom number 0 is known and the modification becomes

$$\tilde{A}^{(0)} = A = \frac{1}{h}\begin{pmatrix} h & 0 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} 0 \\ h \end{pmatrix}. \tag{70}$$

In the last cell we set

$$\tilde{A}^{(N_e)} = A = \frac{1}{h}\begin{pmatrix} 1 & -1 \\ 0 & h \end{pmatrix}, \quad \tilde{b}^{(N_e)} = \begin{pmatrix} h \\ D \end{pmatrix}. \tag{71}$$

We can also perform the symmetric modification. This operation affects only the last cell with a nonzero Dirichlet condition. The algorithm is the same as for the global linear system, resulting in

$$\tilde{A}^{(N_e)} = A = \frac{1}{h}\begin{pmatrix} 1 & 0 \\ 0 & h \end{pmatrix}, \quad \tilde{b}^{(N_e)} = \begin{pmatrix} h + D/h \\ D \end{pmatrix}. \tag{72}$$

The reader is encouraged to assemble the element matrices and vectors and check that the result coincides with the system (69).

5 Boundary conditions: specified derivative

Suppose our model problem $-u''(x) = f(x)$ features the boundary conditions $u'(0) = C$ and $u(L) = D$. As already indicated in Section 2, the former condition can be incorporated through the boundary term that arises from integration by parts. The details of this method will now be illustrated in the context of finite element basis functions.

5.1 The variational formulation

Starting with the Galerkin method,

$$\int_0^L (u''(x) + f(x))\psi_i(x)\,\mathrm{d}x = 0, \quad i \in \mathcal{I}_s,$$

integrating $u''\psi_i$ by parts results in

$$\int_0^L u'(x)'\psi_i'(x)\,\mathrm{d}x - (u'(L)\psi_i(L) - u'(0)\psi_i(0)) = \int_0^L f(x)\psi_i(x)\,\mathrm{d}x, \quad i \in \mathcal{I}_s\,.$$

The first boundary term, $u'(L)\psi_i(L)$, vanishes because $u(L) = D$. The second boundary term, $u'(0)\psi_i(0)$, can be used to implement the condition $u'(0) = C$, provided $\psi_i(0) \neq 0$ for some i (but with finite elements we fortunately have $\psi_0(0) = 1$). The variational form of the differential equation then becomes

$$\int_0^L u'(x)\varphi_i'(x)\,\mathrm{d}x + C\varphi_i(0) = \int_0^L f(x)\varphi_i(x)\,\mathrm{d}x, \quad i \in \mathcal{I}_s\,.$$

5.2 Boundary term vanishes because of the test functions

At points where u is known we may require ψ_i to vanish. Here, $u(L) = D$ and then $\psi_i(L) = 0$, $i \in \mathcal{I}_s$. Obviously, the boundary term $u'(L)\psi_i(L)$ then vanishes.

The set of basis functions $\{\psi_i\}_{i\in\mathcal{I}_s}$ contains, in this case, all the finite element basis functions on the mesh, except the one that is 1 at $x = L$. The basis function

that is left out is used in a boundary function $B(x)$ instead. With a left-to-right numbering, $\psi_i = \varphi_i$, $i = 0, \ldots, N_n - 2$, and $B(x) = D\varphi_{N_n-1}$:

$$u(x) = D\varphi_{N_n-1}(x) + \sum_{j=0}^{N=N_n-2} c_j\varphi_j(x)\,.$$

Inserting this expansion for u in the variational form (5.1) leads to the linear system

$$\sum_{j=0}^{N}\left(\int_0^L \varphi_i'(x)\varphi_j'(x)\,\mathrm{d}x\right)c_j = \int_0^L \left(f(x)\varphi_i(x) - D\varphi_{N_n-1}'(x)\varphi_i'(x)\right)\,\mathrm{d}x - C\varphi_i(0), \tag{73}$$

for $i = 0, \ldots, N = N_n - 2$.

5.3 Boundary term vanishes because of linear system modifications

We may, as an alternative to the approach in the previous section, use a basis $\{\psi_i\}_{i\in\mathcal{I}_s}$ which contains all the finite element functions on the mesh: $\psi_i = \varphi_i$, $i = 0, \ldots, N_n - 1 = N$. In this case, $u'(L)\psi_i(L) = u'(L)\varphi_i(L) \neq 0$ for the i corresponding to the boundary node at $x = L$ (where $\varphi_i = 1$). The number of this node is $i = N_n - 1 = N$ if a left-to-right numbering of nodes is utilized.

However, even though $u'(L)\varphi_{N_n-1}(L) \neq 0$, we do not need to compute this term. For $i < N_n - 1$ we realize that $\varphi_i(L) = 0$. The only nonzero contribution to the right-hand side comes from $i = N$ (b_N). Without a boundary function we must implement the condition $u(L) = D$ by the equivalent statement $c_N = D$ and modify the linear system accordingly. This modification will erase the last row and replace b_N by another value. Any attempt to compute the boundary term $u'(L)\varphi_{N_n-1}(L)$ and store it in b_N will be lost. Therefore, we can safely forget about boundary terms corresponding to Dirichlet boundary conditions also when we use the methods from Section 4.3 or Section 4.4.

The expansion for u reads

$$u(x) = \sum_{j\in\mathcal{I}_s} c_j\varphi_j(x)\,.$$

Insertion in the variational form (5.1) leads to the linear system

$$\sum_{j\in\mathcal{I}_s}\left(\int_0^L \varphi_i'(x)\varphi_j'(x)\,\mathrm{d}x\right)c_j = \int_0^L \left(f(x)\varphi_i(x)\right)\,\mathrm{d}x - C\varphi_i(0), \quad i\in\mathcal{I}_s\,. \tag{74}$$

After having computed the system, we replace the last row by $c_N = D$, either straightforwardly as in Section 4.3 or in a symmetric fashion as in Section 4.4. These modifications can also be performed in the element matrix and vector for the right-most cell.

5.4 Direct computation of the global linear system

We now turn to actual computations with P1 finite elements. The focus is on how the linear system and the element matrices and vectors are modified by the condition $u'(0) = C$.

Consider first the approach where Dirichlet conditions are incorporated by a $B(x)$ function and the known degree of freedom C_{N_n-1} is left out from the linear system (see Section 5.2). The relevant formula for the linear system is given by (73). There are three differences compared to the extensively computed case where $u(0) = 0$ in Sections 3.2 and 3.4. First, because we do not have a Dirichlet condition at the left boundary, we need to extend the linear system (56) with an equation associated with the node $x_0 = 0$. According to Section 4.3, this extension consists of including $A_{0,0} = 1/h$, $A_{0,1} = -1/h$, and $b_0 = h$. For $i > 0$ we have $A_{i,i} = 2/h$, $A_{i-1,i} = A_{i,i+1} = -1/h$. Second, we need to include the extra term $-C\varphi_i(0)$ on the right-hand side. Since all $\varphi_i(0) = 0$ for $i = 1, \ldots, N$, this term reduces to $-C\varphi_0(0) = -C$ and affects only the first equation ($i = 0$). We simply add $-C$ to b_0 such that $b_0 = h - C$. Third, the boundary term $-\int_0^L D\varphi_{N_n-1}(x)\varphi_i \,\mathrm{d}x$ must be computed. Since $i = 0, \ldots, N = N_n - 2$, this integral can only get a nonzero contribution with $i = N_n - 2$ over the last cell. The result becomes $-Dh/6$. The resulting linear system can be summarized in the form

$$\frac{1}{h}\begin{pmatrix} 1 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} h - C \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h - Dh/6 \end{pmatrix}. \tag{75}$$

Next we consider the technique where we modify the linear system to incorporate Dirichlet conditions (see Section 5.3). Now $N = N_n - 1$. The two differences from the case above is that the $-\int_0^L D\varphi_{N_n-1}\varphi_i \,\mathrm{d}x$ term is left out of the right-hand side and an extra last row associated with the node $x_{N_n-1} = L$ where the Dirichlet condition applies is appended to the system. This last row is anyway replaced by the condition $c_N = D$ or this condition can be incorporated in a symmetric fashion. Using the simplest, former approach gives

$$\frac{1}{h}\begin{pmatrix} 1 & -1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 2 & -1 & \ddots & & & & & \vdots \\ 0 & -1 & 2 & -1 & \ddots & & & & \vdots \\ \vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & 0 & -1 & 2 & -1 & \ddots & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & & \ddots & -1 & 2 & -1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 & h \end{pmatrix}\begin{pmatrix} c_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} h-C \\ 2h \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 2h \\ D \end{pmatrix}. \tag{76}$$

5.5 Cellwise computations

Now we compute with one element at a time, working in the reference coordinate system $X \in [-1,1]$. We need to see how the $u'(0) = C$ condition affects the element matrix and vector. The extra term $-C\varphi_i(0)$ in the variational formulation only affects the element vector in the first cell. On the reference cell, $-C\varphi_i(0)$ is transformed to $-C\tilde{\varphi}_r(-1)$, where r counts local degrees of freedom. We have $\tilde{\varphi}_0(-1) = 1$ and $\tilde{\varphi}_1(-1) = 0$ so we are left with the contribution $-C\tilde{\varphi}_0(-1) = -C$ to $\tilde{b}_0^{(0)}$:

$$\tilde{A}^{(0)} = A = \frac{1}{h}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(0)} = \begin{pmatrix} h-C \\ h \end{pmatrix}. \tag{77}$$

No other element matrices or vectors are affected by the $-C\varphi_i(0)$ boundary term.

There are two alternative ways of incorporating the Dirichlet condition. Following Section 5.2, we get a 1×1 element matrix in the last cell and an element vector with an extra term containing D:

$$\tilde{A}^{(e)} = \frac{1}{h}\begin{pmatrix} 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h\begin{pmatrix} 1 - D/6 \end{pmatrix}, \tag{78}$$

Alternatively, we include the degree of freedom at the node with u specified. The element matrix and vector must then be modified to constrain the $\tilde{c}_1 = c_N$ value at local node $r = 1$:

$$\tilde{A}^{(N_e)} = A = \frac{1}{h}\begin{pmatrix} 1 & 1 \\ 0 & h \end{pmatrix}, \quad \tilde{b}^{(N_e)} = \begin{pmatrix} h \\ D \end{pmatrix}. \tag{79}$$

6 Implementation

At this point, it is sensible to create a program with symbolic calculations to perform all the steps in the computational machinery, both for automating the work and for documenting the complete algorithms. As we have seen, there are quite many details involved with finite element computations and incorporation of boundary conditions. An implementation will also act as a structured summary of all these details.

6.1 Global basis functions

We first consider implementations when ψ_i are global functions are hence different from zero on most of $\Omega = [0, L]$ so all integrals need integration over the entire domain. (Finite element basis functions, where we utilize their local support and perform integrations over cells, will be treated later.) Since the expressions for the entries in the linear system depend on the differential equation problem being solved, the user must supply the necessary formulas via Python functions. The implementations here attempt to perform symbolic calculations, but fall back on numerical computations if the symbolic ones fail.

The user must prepare a function `integrand_lhs(psi, i, j)` for returning the integrand of the integral that contributes to matrix entry (i, j). The `psi` variable is a Python dictionary holding the basis functions and their derivatives in symbolic form. More precisely, `psi[q]` is a list of

$$\{\frac{d^q\psi_0}{dx^q}, \ldots, \frac{d^q\psi_{N_n-1}}{dx^q}\}.$$

Similarly, `integrand_rhs(psi, i)` returns the integrand for entry number i in the right-hand side vector.

Since we also have contributions to the right-hand side vector (and potentially also the matrix) from boundary terms without any integral, we introduce two additional functions, `boundary_lhs(psi, i, j)` and `boundary_rhs(psi, i)` for returning terms in the variational formulation that are not to be integrated over the domain Ω. Examples, to be shown later, will explain in more detail how these user-supplied function may look like.

The linear system can be computed and solved symbolically by the following function:

```
import sympy as sym

def solver(integrand_lhs, integrand_rhs, psi, Omega,
           boundary_lhs=None, boundary_rhs=None):
    N = len(psi[0]) - 1
    A = sym.zeros((N+1, N+1))
    b = sym.zeros((N+1, 1))
    x = sym.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = integrand_lhs(psi, i, j)
            I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
            if boundary_lhs is not None:
```

```
                I += boundary_lhs(psi, i, j)
            A[i,j] = A[j,i] = I   # assume symmetry
        integrand = integrand_rhs(psi, i)
        I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
        if boundary_rhs is not None:
            I += boundary_rhs(psi, i)
        b[i,0] = I
    c = A.LUsolve(b)
    u = sum(c[i,0]*psi[0][i] for i in range(len(psi[0])))
    return u, c
```

Not surprisingly, symbolic solution of differential equations, discretized by a Galerkin or least squares method with global basis functions, is of limited interest beyond the simplest problems, because symbolic integration might be very time consuming or impossible, not only in `sympy` but also in WolframAlpha[4] (which applies the perhaps most powerful symbolic integration software available today: Mathematica). Numerical integration as an option is therefore desirable.

The extended `solver` function below tries to combine symbolic and numerical integration. The latter can be enforced by the user, or it can be invoked after a non-successful symbolic integration (being detected by an `Integral` object as the result of the integration in `sympy`). Note that for a numerical integration, symbolic expressions must be converted to Python functions (using `lambdify`), and the expressions cannot contain other symbols than `x`. The real `solver` routine in the `varform1D.py`[5] file has error checking and meaningful error messages in such cases. The `solver` code below is a condensed version of the real one, with the purpose of showing how to automate the Galerkin or least squares method for solving differential equations in 1D with global basis functions:

```
def solver(integrand_lhs, integrand_rhs, psi, Omega,
           boundary_lhs=None, boundary_rhs=None, symbolic=True):
    N = len(psi[0]) - 1
    A = sym.zeros((N+1, N+1))
    b = sym.zeros((N+1, 1))
    x = sym.Symbol('x')
    for i in range(N+1):
        for j in range(i, N+1):
            integrand = integrand_lhs(psi, i, j)
            if symbolic:
                I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
                if isinstance(I, sym.Integral):
                    symbolic = False  # force num.int. hereafter
            if not symbolic:
                integrand = sym.lambdify([x], integrand)
                I = sym.mpmath.quad(integrand, [Omega[0], Omega[1]])
            if boundary_lhs is not None:
                I += boundary_lhs(psi, i, j)
            A[i,j] = A[j,i] = I
        integrand = integrand_rhs(psi, i)
        if symbolic:
            I = sym.integrate(integrand, (x, Omega[0], Omega[1]))
            if isinstance(I, sym.Integral):
                symbolic = False
        if not symbolic:
```

[4] `http://wolframalpha.com`

[5] `http://tinyurl.com/nm5587k/varform/varform1D.py`

```
            integrand = sym.lambdify([x], integrand)
            I = sym.mpmath.quad(integrand, [Omega[0], Omega[1]])
        if boundary_rhs is not None:
            I += boundary_rhs(psi, i)
        b[i,0] = I
    c = A.LUsolve(b)
    u = sum(c[i,0]*psi[0][i] for i in range(len(psi[0])))
    return u, c
```

6.2 Example: constant right-hand side

To demonstrate the code above, we address

$$-u''(x) = b, \quad x \in \Omega = [0,1], \quad u(0) = 1,\ u(1) = 0,$$

with b as a (symbolic) constant. A possible basis for the space V is $\psi_i(x) = x^{i+1}(1-x)$, $i \in \mathcal{I}_s$. Note that $\psi_i(0) = \psi_i(1) = 0$ as required by the Dirichlet conditions. We need a $B(x)$ function to take care of the known boundary values of u. Any function $B(x) = 1 - x^p$, $p \in \mathbb{R}$, is a candidate, and one arbitrary choice from this family is $B(x) = 1 - x^3$. The unknown function is then written as

$$u(x) = B(x) + \sum_{j \in \mathcal{I}_s} c_j \psi_j(x)\,.$$

Let us use the Galerkin method to derive the variational formulation. Multiplying the differential equation by v and integrating by parts yield

$$\int_0^1 u'v'\,\mathrm{d}x = \int_0^1 fv\,\mathrm{d}x \quad \forall v \in V,$$

and with $u = B + \sum_j c_j\psi_j$ we get the linear system

$$\sum_{j \in \mathcal{I}_s} \left(\int_0^1 \psi_i'\psi_j'\,\mathrm{d}x \right) c_j = \int_0^1 (f\psi_i - B'\psi_i')\,\mathrm{d}x, \quad i \in \mathcal{I}_s\,. \tag{80}$$

The application can be coded as follows with `sympy`:

```
import sympy as sym
x, b = sym.symbols('x b')
f = b
B = 1 - x**3
dBdx = sym.diff(B, x)

# Compute basis functions and their derivatives
N = 3
psi = {0: [x**(i+1)*(1-x) for i in range(N+1)]}
psi[1] = [sym.diff(psi_i, x) for psi_i in psi[0]]

def integrand_lhs(psi, i, j):
    return psi[1][i]*psi[1][j]

def integrand_rhs(psi, i):
```

```
    return f*psi[0][i] - dBdx*psi[1][i]

Omega = [0, 1]

from varform1D import solver
u_bar, c = solver(integrand_lhs, integrand_rhs, psi, Omega,
                  verbose=True, symbolic=True)
u = B + u_bar
print 'solution u:', sym.simplify(sym.expand(u))
```

The printout of `u` reads `-b*x**2/2 + b*x/2 - x + 1`. Note that expanding `u`, before simplifying, is necessary in the present case to get a compact, final expression with `sympy`. Doing `expand` before `simplify` is a common strategy for simplifying expressions in `sympy`. However, a non-expanded `u` might be preferable in other cases - this depends on the problem in question.

The exact solution $u_{\mathrm{e}}(x)$ can be derived by some `sympy` code that closely follows the examples in Section 1.2. The idea is to integrate $-u'' = b$ twice and determine the integration constants from the boundary conditions:

```
C1, C2 = sym.symbols('C1 C2')    # integration constants
f1 = sym.integrate(f, x) + C1
f2 = sym.integrate(f1, x) + C2
# Find C1 and C2 from the boundary conditions u(0)=0, u(1)=1
s = sym.solve([u_e.subs(x,0) - 1, u_e.subs(x,1) - 0], [C1, C2])
# Form the exact solution
u_e = -f2 + s[C1]*x + s[C2]
print 'analytical solution:', u_e
print 'error:', sym.simplify(sym.expand(u - u_e))
```

The last line prints `0`, which is not surprising when $u_{\mathrm{e}}(x)$ is a parabola and our approximate u contains polynomials up to degree 4. It suffices to have $N = 1$, i.e., polynomials of degree 2, to recover the exact solution.

We can play around with the code and test that with $f = Kx^p$, for some constants K and p, the solution is a polynomial of degree $p + 2$, and $N = p + 1$ guarantees that the approximate solution is exact.

Although the symbolic code is capable of integrating many choices of $f(x)$, the symbolic expressions for u quickly become lengthy and non-informative, so numerical integration in the code, and hence numerical answers, have the greatest application potential.

6.3 Finite elements

Implementation of the finite element algorithms for differential equations follows closely the algorithm for approximation of functions. The new additional ingredients are

1. other types of integrands (as implied by the variational formulation)
2. additional boundary terms in the variational formulation for Neumann boundary conditions

3. modification of element matrices and vectors due to Dirichlet boundary conditions

Point 1 and 2 can be taken care of by letting the user supply functions defining the integrands and boundary terms on the left- and right-hand side of the equation system:

- Integrand on the left-hand side: `ilhs(e, phi, r, s, X, x, h)`
- Integrand on the right-hand side: `irhs(e, phi, r, X, x, h)`
- Boundary term on the left-hand side: `blhs (e, phi, r, s, X, x, h)`
- Boundary term on the right-hand side: `brhs (e, phi, r, s, X, x, h)`

Here, `phi` is a dictionary where `phi[q]` holds a list of the derivatives of order `q` of the basis functions with respect to the physical coordinate x. The derivatives are available as Python functions of `X`. For example, `phi[0][r](X)` means $\tilde{\varphi}_r(X)$, and `phi[1][s](X, h)` means $d\tilde{\varphi}_s(X)/dx$ (we refer to the file `fe1D.py`[6] for details regarding the function `basis` that computes the `phi` dictionary). The `r` and `s` arguments in the above functions correspond to the index in the integrand contribution from an integration point to $\tilde{A}^{(e)}_{r,s}$ and $\tilde{b}^{(e)}_r$. The variables `e` and `h` are the current element number and the length of the cell, respectively. Specific examples below will make it clear how to construction these Python functions.

Given a mesh represented by `vertices`, `cells`, and `dof_map` as explained before, we can write a pseudo Python code to list all the steps in the computational algorithm for finite element solution of a differential equation.

```
<Declare global matrix and rhs: A, b>

for e in range(len(cells)):

    # Compute element matrix and vector
    n = len(dof_map[e])  # no of dofs in this element
    h = vertices[cells[e][1]] - vertices[cells[e][1]]
    <Initialize element matrix and vector: A_e, b_e>

    # Integrate over the reference cell
    points, weights = <numerical integration rule>
    for X, w in zip(points, weights):
        phi = <basis functions and derivatives at X>
        detJ = h/2
        dX = detJ*w

        x = <affine mapping from X>
        for r in range(n):
            for s in range(n):
                A_e[r,s] += ilhs(e, phi, r, s, X, x, h)*dX
            b_e[r] += irhs(e, phi, r, X, x, h)*dX

    # Add boundary terms
    for r in range(n):
        for s in range(n):
```

[6] `http://tinyurl.com/nm5587k/varform/fe1D.py`

```
            A_e[r,s] += blhs(e, phi, r, s, X, x)*dX
        b_e[r] += brhs(e, phi, r, X, x, h)*dX

    # Incorporate essential boundary conditions
    for r in range(n):
        global_dof = dof_map[e][r]
        if global_dof in essbc:
            # local dof r is subject to an essential condition
            value = essbc[global_dof]
            # Symmetric modification
            b_e -= value*A_e[:,r]
            A_e[r,:] = 0
            A_e[:,r] = 0
            A_e[r,r] = 1
            b_e[r] = value

    # Assemble
    for r in range(n):
        for s in range(n):
            A[dof_map[e][r], dof_map[e][s]] += A_e[r,s]
        b[dof_map[e][r] += b_e[r]

<solve linear system>
```

A complete function `finite_element1D_naive` for the 1D finite algorithm above, is found in the file `fe1D.py`[7]. The term "naive" refers to a version of the algorithm where we use a standard dense square matrix as global matrix `A`. The implementation also has a verbose mode for printing out the element matrices and vectors as they are computed. Below is the complete function without the print statements.

```
def finite_element1D_naive(
    vertices, cells, dof_map,     # mesh
    essbc,                        # essbc[globdof]=value
    ilhs,                         # integrand left-hand side
    irhs,                         # integrand right-hand side
    blhs=lambda e, phi, r, s, X, x, h: 0,
    brhs=lambda e, phi, r, X, x, h: 0,
    intrule='GaussLegendre',      # integration rule class
    verbose=False,                # print intermediate results?
    ):
    N_e = len(cells)
    N_n = np.array(dof_map).max() + 1

    A = np.zeros((N_n, N_n))
    b = np.zeros(N_n)

    for e in range(N_e):
        Omega_e = [vertices[cells[e][0]], vertices[cells[e][1]]]
        h = Omega_e[1] - Omega_e[0]

        d = len(dof_map[e]) - 1  # Polynomial degree
        # Compute all element basis functions and their derivatives
        phi = basis(d)

        # Element matrix and vector
        n = d+1  # No of dofs per element
        A_e = np.zeros((n, n))
```

[7] `http://tinyurl.com/nm5587k/varform/fe1D.py`

```
        b_e = np.zeros(n)

        # Integrate over the reference cell
        if intrule == 'GaussLegendre':
            points, weights = GaussLegendre(d+1)
        elif intrule == 'NewtonCotes':
            points, weights = NewtonCotes(d+1)

        for X, w in zip(points, weights):
            detJ = h/2
            x = affine_mapping(X, Omega_e)
            dX = detJ*w

            # Compute contribution to element matrix and vector
            for r in range(n):
                for s in range(n):
                    A_e[r,s] += ilhs(phi, r, s, X, x, h)*dX
                b_e[r] += irhs(phi, r, X, x, h)*dX

        # Add boundary terms
        for r in range(n):
            for s in range(n):
                A_e[r,s] += blhs(phi, r, s, X, x, h)
            b_e[r] += brhs(phi, r, X, x, h)

        # Incorporate essential boundary conditions
        modified = False
        for r in range(n):
            global_dof = dof_map[e][r]
            if global_dof in essbc:
                # dof r is subject to an essential condition
                value = essbc[global_dof]
                # Symmetric modification
                b_e -= value*A_e[:,r]
                A_e[r,:] = 0
                A_e[:,r] = 0
                A_e[r,r] = 1
                b_e[r] = value
                modified = True

        # Assemble
        for r in range(n):
            for s in range(n):
                A[dof_map[e][r], dof_map[e][s]] += A_e[r,s]
            b[dof_map[e][r]] += b_e[r]

    c = np.linalg.solve(A, b)
    return c, A, b, timing
```

The `timing` object is a dictionary holding the CPU spent on computing `A` and the CPU time spent on solving the linear system. (We have left out the timing statements.)

6.4 Utilizing a sparse matrix

A potential efficiency problem with the `finite_element1D_naive` function is that it uses dense $(N+1) \times (N+1)$ matrices, while we know that only $2d+1$ diagonals around the main diagonal are different from zero. Switching to a sparse matrix is very easy. Using the DOK (dictionary of keys) format, we declare `A` as

```
import scipy.sparse
A = scipy.sparse.dok_matrix((N_n, N_n))
```

Assignments or in-place arithmetics are done as for a dense matrix,

```
A[i,j] += term
A[i,j]  = term
```

but only the index pairs `(i,j)` we have used in assignments or in-place arithmetics are actually stored. A tailored solution algorithm is needed. The most reliable is sparse Gaussian elimination:

```
import scipy.sparse.linalg
c = scipy.sparse.linalg.spsolve(A.tocsr(), b, use_umfpack=True)
```

The declaration of `A` and the solve statement are the only changes needed in the `finite_element1D_naive` to utilize sparse matrices. The resulting modification is found in the function `finite_element1D`.

Example. Let us demonstrate the finite element software on

$$-u''(x) = f(x), \quad x \in (0, L), \quad u'(0) = C,\ u(L) = D\,.$$

This problem can be analytically solved by the `model2` function from Section 1.2. Let $f(x) = x^2$. Calling `model2(x**2, L, C, D)` gives

$$u(x) = D + C(x - L) + \frac{1}{12}(L^4 - x^4)$$

The variational formulation reads

$$(u', v) = (x^2, v) - Cv(0)\,.$$

The entries in the element matrix and vector, which we need to set up the `ilhs`, `irhs`, `blhs`, and `brhs` functions, becomes

$$A^{(e)}_{r,s} = \int_{-1}^{1} \frac{d\tilde{\varphi}_r}{dx}\frac{\tilde{\varphi}_s}{dx}(\det J\,\mathrm{d}X),$$
$$b^{(e)} = \int_{-1}^{1} x^2\tilde{\varphi}_r \det J\,\mathrm{d}X - C\tilde{\varphi}_r(-1)I(e,0),$$

where $I(e)$ is an indicator function: $I(e,q) = 1$ if $e = q$, otherwise $I(e) = 0$. We use this indicator function to formulate that the boundary term $Cv(0)$, which in the local element coordinate system becomes $C\tilde{\varphi}_r(-1)$, is only included for the element $e = 0$.

The functions for specifying the element matrix and vector entries must contain the integrand, but without the $\det J\,\mathrm{d}X$ term, and the derivatives $d\tilde{\varphi}_r(X)/dx$ with respect to the physical x coordinates are contained in `phi[1][r](X)`, computed by the function `basis`.

```
def ilhs(e, phi, r, s, X, x, h):
    return phi[1][r](X, h)*phi[1][s](X, h)

def irhs(e, phi, r, X, x, h):
    return x**2*phi[0][r](X)

def blhs(e, phi, r, s, X, x, h):
    return 0

def brhs(e, phi, r, X, x, h):
    return -C*phi[0][r](-1) if e == 0 else 0
```

We can then make the call to `finite_element1D_naive` or `finite_element1D` to solve the problem with two P1 elements:

```
from fe1D import finite_element1D_naive, mesh_uniform
C = 5;  D = 2;  L = 4
d = 1

vertices, cells, dof_map = mesh_uniform(
    N_e=2, d=d, Omega=[0,L], symbolic=False)
essbc = {}
essbc[dof_map[-1][-1]] = D

c, A, b, timing = finite_element1D(
    vertices, cells, dof_map, essbc,
    ilhs=ilhs, irhs=irhs, blhs=blhs, brhs=brhs,
    intrule='GaussLegendre')
```

It remains to plot the solution (with high resolution in each element). To this end, we use the `u_glob` function imported from `fe1D`, which imports it from `fe_approx1D_numit` (the `u_glob` function in `fe_approx1D.py` works with `elements` and `nodes`, while `u_glob` in `fe_approx1D_numint` works with `cells`, `vertices`, and `dof_map`):

```
u_exact = lambda x: D + C*(x-L) + (1./6)*(L**3 - x**3)
from fe1D import u_glob
x, u, nodes = u_glob(c, cells, vertices, dof_map)
u_e = u_exact(x, C, D, L)
print u_exact(nodes, C, D, L) - c  # difference at the nodes

import matplotlib.pyplot as plt
plt.plot(x, u, 'b-', x, u_e, 'r--')
plt.legend(['finite elements, d=%d' %d, 'exact'], loc='upper left')
plt.show()
```

The result is shown in Figure 3. We see that the solution using P1 elements is exact at the nodes, but feature considerable discrepancy between the nodes. Exercise 10 asks you to explore this problem further using other m and d values.

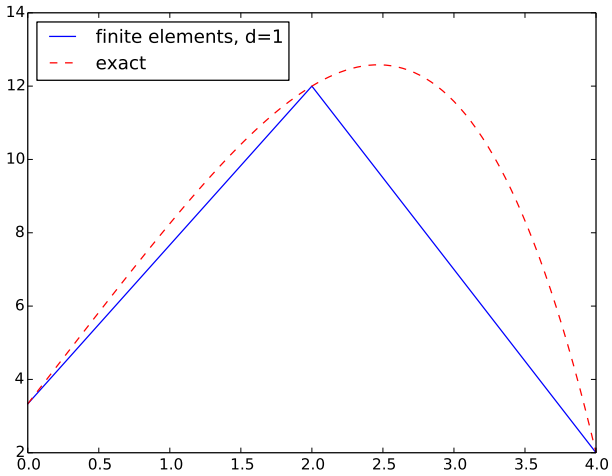


Figure 3: Finite element and exact solution using two cells.

7 Variational formulations in 2D and 3D

The major difference between deriving variational formulations in 2D and 3D compared to 1D is the rule for integrating by parts. The cells have shapes different from an interval, so basis functions look a bit different, and there is a technical difference in actually calculating the integrals over cells. Otherwise, going to 2D and 3D is not a big step from 1D. All the fundamental ideas still apply.

7.1 Integration by parts

A typical second-order term in a PDE may be written in dimension-independent notation as

$$\nabla^2 u \quad \text{or} \quad \nabla\cdot(\alpha(\boldsymbol{x})\nabla u)\ .$$

The explicit forms in a 2D problem become

$$\nabla^2 u = \nabla\cdot\nabla u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

and

$$\nabla\cdot(a(\boldsymbol{x})\nabla u) = \frac{\partial}{\partial x}\left(\alpha(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\alpha(x,y)\frac{\partial u}{\partial y}\right)\ .$$

We shall continue with the latter operator as the former arises from just setting $\alpha = 1$.

The integration by parts formula for $\int \nabla\cdot(\alpha\nabla)$.

The general rule for integrating by parts is often referred to as Green's first identity[a]:

$$-\int_\Omega \nabla\cdot(\alpha(\boldsymbol{x})\nabla u)v\,\mathrm{d}x = \int_\Omega \alpha(\boldsymbol{x})\nabla u\cdot\nabla v\,\mathrm{d}x - \int_{\partial\Omega} a\frac{\partial u}{\partial n}v\,\mathrm{d}s, \tag{81}$$

where $\partial\Omega$ is the boundary of Ω and $\partial u/\partial n = \boldsymbol{n}\cdot\nabla u$ is the derivative of u in the outward normal direction, $\boldsymbol{n}$ being an outward unit normal to $\partial\Omega$. The integrals $\int_\Omega()\,\mathrm{d}x$ are area integrals in 2D and volume integrals in 3D, while $\int_{\partial\Omega}()\,\mathrm{d}s$ is a line integral in 2D and a surface integral in 3D.

[a] `http://en.wikipedia.org/wiki/Green's_identities`

It will be convenient to divide the boundary into two parts:

- $\partial\Omega_N$, where we have Neumann conditions $-a\frac{\partial u}{\partial n} = g$, and
- $\partial\Omega_D$, where we have Dirichlet conditions $u = u_0$.

The test functions v are (as usual) required to vanish on $\partial\Omega_D$.

7.2 Example on a multi-dimensional variational problem

Here is a quite general, stationary, linear PDE arising in many problems:

$$\boldsymbol{v}\cdot\nabla u + \beta u = \nabla\cdot(\alpha\nabla u) + f, \quad \boldsymbol{x}\in\Omega, \tag{82}$$

$$u = u_0, \quad \boldsymbol{x}\in\partial\Omega_D, \tag{83}$$

$$-\alpha\frac{\partial u}{\partial n} = g, \quad \boldsymbol{x}\in\partial\Omega_N\,. \tag{84}$$

The vector field $\boldsymbol{v}$ and the scalar functions a, α, f, u_0, and g may vary with the spatial coordinate $\boldsymbol{x}$ and must be known.

Such a second-order PDE needs exactly one boundary condition at each point of the boundary, so $\partial\Omega_N\cup\partial\Omega_D$ must be the complete boundary $\partial\Omega$.

Assume that the boundary function $u_0(\boldsymbol{x})$ is defined for all $\boldsymbol{x}\in\Omega$. The unknown function can then be expanded as

$$u = B + \sum_{j\in\mathcal{I}_s} c_j\psi_j, \quad B = u_0\,.$$

As long as any $\psi_j = 0$ on $\partial\Omega_D$, we realize that $u = u_0$ on $\partial\Omega_D$.

The variational formula is obtained from Galerkin's method, which technically means multiplying the PDE by a test function v and integrating over Ω:

$$\int_\Omega (\boldsymbol{v}\cdot\nabla u + \beta u)v\,\mathrm{d}x = \int_\Omega \nabla\cdot(\alpha\nabla u)\,\mathrm{d}x + \int_\Omega fv\,\mathrm{d}x\,.$$

The second-order term is integrated by parts, according to the formula (81):

$$\int_\Omega \nabla\cdot(\alpha\nabla u)\,v\,\mathrm{d}x = -\int_\Omega \alpha\nabla u\cdot\nabla v\,\mathrm{d}x + \int_{\partial\Omega}\alpha\frac{\partial u}{\partial n}v\,\mathrm{d}s\,.$$

Galerkin's method therefore leads to

$$\int_\Omega (\boldsymbol{v}\cdot\nabla u + \beta u)v\,\mathrm{d}x = -\int_\Omega \alpha\nabla u\cdot\nabla v\,\mathrm{d}x + \int_{\partial\Omega}\alpha\frac{\partial u}{\partial n}v\,\mathrm{d}s + \int_\Omega fv\,\mathrm{d}x\,.$$

The boundary term can be developed further by noticing that $v\neq 0$ only on $\partial\Omega_N$,

$$\int_{\partial\Omega}\alpha\frac{\partial u}{\partial n}v\,\mathrm{d}s = \int_{\partial\Omega_N}\alpha\frac{\partial u}{\partial n}v\,\mathrm{d}s,$$

and that on $\partial\Omega_N$, we have the condition $a\frac{\partial u}{\partial n} = -g$, so the term becomes

$$-\int_{\partial\Omega_N} gv\,\mathrm{d}s\,.$$

The final variational form is then

$$\int_\Omega (\boldsymbol{v}\cdot\nabla u + \beta u)v\,\mathrm{d}x = -\int_\Omega \alpha\nabla u\cdot\nabla v\,\mathrm{d}x - \int_{\partial\Omega_N} gv\,\mathrm{d}s + \int_\Omega fv\,\mathrm{d}x\,.$$

Instead of using the integral signs, we may use the inner product notation:

$$(\boldsymbol{v}\cdot\nabla u, v) + (\beta u, v) = -(\alpha\nabla u, \nabla v) - (g, v)_N + (f, v)\,.$$

The subscript $_N$ in $(g,v)_N$ is a notation for a line or surface integral over $\partial\Omega_N$, while $(\cdot,\cdot)$ is the area/volume integral over Ω.

We can derive explicit expressions for the linear system for $\{c_j\}_{j\in\mathcal{I}_s}$ that arises from the variational formulation. Inserting the u expansion results in

$$\sum_{j\in\mathcal{I}_s}((\boldsymbol{v}\cdot\nabla\psi_j,\psi_i) + (\beta\psi_j,\psi_i) + (\alpha\nabla\psi_j,\nabla\psi_i))c_j = (g,\psi_i)_N + (f,\psi_i) - (\boldsymbol{v}\cdot\nabla u_0,\psi_i) + (\beta u_0,\psi_i) + (\alpha\nabla u_0,\nabla\psi_i)\,.$$

This is a linear system with matrix entries

$$A_{i,j} = (\boldsymbol{v}\cdot\nabla\psi_j,\psi_i) + (\beta\psi_j,\psi_i) + (\alpha\nabla\psi_j,\nabla\psi_i)$$

and right-hand side entries

$$b_i = (g, \psi_i)_N + (f, \psi_i) - (\boldsymbol{v} \cdot \nabla u_0, \psi_i) + (\beta u_0, \psi_i) + (\alpha \nabla u_0, \nabla \psi_i),$$

for $i, j \in \mathcal{I}_s$.

In the finite element method, we usually express u_0 in terms of basis functions and restrict i and j to run over the degrees of freedom that are not prescribed as Dirichlet conditions. However, we can also keep all the $\{c_j\}_{j\in\mathcal{I}_s}$ as unknowns, drop the u_0 in the expansion for u, and incorporate all the known c_j values in the linear system. This has been explained in detail in the 1D case, and the technique is the same for 2D and 3D problems.

7.3 Transformation to a reference cell in 2D and 3D

The real power of the finite element method first becomes evident when we want to solve partial differential equations posed on two- and three-dimensional domains of non-trivial geometric shape. As in 1D, the domain Ω is divided into N_e non-overlapping cells. The elements have simple shapes: triangles and quadrilaterals are popular in 2D, while tetrahedra and box-shapes elements dominate in 3D. The finite element basis functions φ_i are, as in 1D, polynomials over each cell. The integrals in the variational formulation are, as in 1D, split into contributions from each cell, and these contributions are calculated by mapping a physical cell, expressed in physical coordinates $\boldsymbol{x}$, to a reference cell in a local coordinate system $\boldsymbol{X}$. This mapping will now be explained in detail.

We consider an integral of the type

$$\int_{\Omega^{(e)}} \alpha(\boldsymbol{x}) \nabla\varphi_i \cdot \nabla\varphi_j \,\mathrm{d}x, \tag{85}$$

where the φ_i functions are finite element basis functions in 2D or 3D, defined in the physical domain. Suppose we want to calculate this integral over a reference cell, denoted by $\tilde{\Omega}^r$, in a coordinate system with coordinates $\boldsymbol{X} = (X_0, X_1)$ (2D) or $\boldsymbol{X} = (X_0, X_1, X_2)$ (3D). The mapping between a point $\boldsymbol{X}$ in the reference coordinate system and the corresponding point $\boldsymbol{x}$ in the physical coordinate system is given by a vector relation $\boldsymbol{x}(\boldsymbol{X})$. The corresponding Jacobian, J, of this mapping has entries

$$J_{i,j} = \frac{\partial x_j}{\partial X_i}\,.$$

The change of variables requires $\mathrm{d}x$ to be replaced by $\det J\,\mathrm{d}X$. The derivatives in the ∇ operator in the variational form are with respect to $\boldsymbol{x}$, which we may denote by $\nabla_{\boldsymbol{x}}$. The $\varphi_i(\boldsymbol{x})$ functions in the integral are replaced by local basis functions $\tilde{\varphi}_r(\boldsymbol{X})$ so the integral features $\nabla_{\boldsymbol{x}}\tilde{\varphi}_r(\boldsymbol{X})$. We readily have $\nabla_{\boldsymbol{X}}\tilde{\varphi}_r(\boldsymbol{X})$ from formulas for the basis functions in the reference cell, but the desired quantity $\nabla_{\boldsymbol{x}}\tilde{\varphi}_r(\boldsymbol{X})$ requires some efforts to compute. All the details are provided below.

Let $i = q(e, r)$ and consider two space dimensions. By the chain rule,

$$\frac{\partial\tilde{\varphi}_r}{\partial X} = \frac{\partial\varphi_i}{\partial X} = \frac{\partial\varphi_i}{\partial x}\frac{\partial x}{\partial X} + \frac{\partial\varphi_i}{\partial y}\frac{\partial y}{\partial X},$$

and

$$\frac{\partial\tilde{\varphi}_r}{\partial Y} = \frac{\partial\varphi_i}{\partial Y} = \frac{\partial\varphi_i}{\partial x}\frac{\partial x}{\partial Y} + \frac{\partial\varphi_i}{\partial y}\frac{\partial y}{\partial Y}.$$

We can write these two equations as a vector equation

$$\begin{bmatrix} \frac{\partial\tilde{\varphi}_r}{\partial X} \\ \frac{\partial\tilde{\varphi}_r}{\partial Y} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial y}{\partial X} \\ \frac{\partial x}{\partial Y} & \frac{\partial y}{\partial Y} \end{bmatrix} \begin{bmatrix} \frac{\partial\varphi_i}{\partial x} \\ \frac{\partial\varphi_i}{\partial y} \end{bmatrix}$$

Identifying

$$\nabla_{\boldsymbol{X}}\tilde{\varphi}_r = \begin{bmatrix} \frac{\partial\tilde{\varphi}_r}{\partial X} \\ \frac{\partial\tilde{\varphi}_r}{\partial Y} \end{bmatrix}, \quad J = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial y}{\partial X} \\ \frac{\partial x}{\partial Y} & \frac{\partial y}{\partial Y} \end{bmatrix}, \quad \nabla_{\boldsymbol{x}}\varphi_r = \begin{bmatrix} \frac{\partial\varphi_i}{\partial x} \\ \frac{\partial\varphi_i}{\partial y} \end{bmatrix},$$

we have the relation

$$\nabla_{\boldsymbol{X}}\tilde{\varphi}_r = J \cdot \nabla_{\boldsymbol{x}}\varphi_i,$$

which we can solve with respect to $\nabla_{\boldsymbol{x}}\varphi_i$:

$$\nabla_{\boldsymbol{x}}\varphi_i = J^{-1} \cdot \nabla_{\boldsymbol{X}}\tilde{\varphi}_r\,. \tag{86}$$

On the reference cell, $\varphi_i(\boldsymbol{x}) = \tilde{\varphi}_r(\boldsymbol{X})$, so

$$\nabla_{\boldsymbol{x}}\tilde{\varphi}_r(\boldsymbol{X}) = J^{-1}(\boldsymbol{X}) \cdot \nabla_{\boldsymbol{X}}\tilde{\varphi}_r(\boldsymbol{X})\,. \tag{87}$$

This means that we have the following transformation of the integral in the physical domain to its counterpart over the reference cell:

$$\int_{\Omega}^{(e)} \alpha(\boldsymbol{x})\nabla_{\boldsymbol{x}}\varphi_i \cdot \nabla_{\boldsymbol{x}}\varphi_j \,\mathrm{d}x \int_{\tilde{\Omega}^r} \alpha(\boldsymbol{x}(\boldsymbol{X}))(J^{-1}\cdot\nabla_{\boldsymbol{X}}\tilde{\varphi}_r)\cdot(J^{-1}\cdot\nabla\tilde{\varphi}_s)\det J\,\mathrm{d}X \tag{88}$$

7.4 Numerical integration

Integrals are normally computed by numerical integration rules. For multi-dimensional cells, various families of rules exist. All of them are similar to what is shown in 1D: $\int f\,\mathrm{d}x \approx \sum_j w_i f(\boldsymbol{x}_j)$, where w_j are weights and $\boldsymbol{x}_j$ are corresponding points.

The file `numint.py`[8] contains the functions `quadrature_for_triangles(n)` and `quadrature_for_tetrahedra(n)`, which returns lists of points and weights corresponding to integration rules with `n` points over the reference triangle with vertices $(0,0)$, $(1,0)$, $(0,1)$, and the reference tetrahedron with vertices $(0,0,0)$, $(1,0,0)$, $(0,1,0)$, $(0,0,1)$, respectively. For example, the first two rules for integration over a triangle have 1 and 3 points:

[8] `http://tinyurl.com/nm5587k/approx/numint.py`

```
>>> import numint
>>> x, w = numint.quadrature_for_triangles(num_points=1)
>>> x
[(0.3333333333333333, 0.3333333333333333)]
>>> w
[0.5]
>>> x, w = numint.quadrature_for_triangles(num_points=3)
>>> x
[(0.16666666666666666, 0.16666666666666666),
 (0.66666666666666666, 0.16666666666666666),
 (0.16666666666666666, 0.66666666666666666)]
>>> w
[0.16666666666666666, 0.16666666666666666, 0.16666666666666666]
```

Rules with 1, 3, 4, and 7 points over the triangle will exactly integrate polynomials of degree 1, 2, 3, and 4, respectively. In 3D, rules with 1, 4, 5, and 11 points over the tetrahedron will exactly integrate polynomials of degree 1, 2, 3, and 4, respectively.

7.5 Convenient formulas for P1 elements in 2D

We shall now provide some formulas for piecewise linear φ_i functions and their integrals *in the physical coordinate system*. These formulas make it convenient to compute with P1 elements without the need to work in the reference coordinate system and deal with mappings and Jacobians. A lot of computational and algorithmic details are hidden by this approach.

Let $\Omega^{(e)}$ be cell number e, and let the three vertices have global vertex numbers I, J, and K. The corresponding coordinates are (x_I, y_I), (x_J, y_J), and (x_K, y_K). The basis function φ_I over $\Omega^{(e)}$ have the explicit formula

$$\varphi_I(x,y) = \frac{1}{2}\Delta\left(\alpha_I + \beta_I x + \gamma_I y\right), \tag{89}$$

where

$$\alpha_I = x_J y_K - x_K y_J, \tag{90}$$
$$\beta_I = y_J - y_K, \tag{91}$$
$$\gamma_I = x_K - x_J,, \tag{92}$$

and

$$2\Delta = \det\begin{pmatrix} 1 & x_I & y_I \\ 1 & x_J & y_J \\ 1 & x_K & y_K \end{pmatrix}. \tag{93}$$

The quantity Δ is the area of the cell.

The following formula is often convenient when computing element matrices and vectors:

$$\int_{\Omega^{(e)}} \varphi_I^p \varphi_J^q \varphi_K^r dxdy = \frac{p!q!r!}{(p+q+r+2)!} 2\Delta\,. \tag{94}$$

(Note that the q in this formula is not to be mixed with the $q(e,r)$ mapping of degrees of freedom.)

As an example, the element matrix entry $\int_{\Omega^{(e)}} \varphi_I \varphi_J \,\mathrm{d}x$ can be computed by setting $p = q = 1$ and $r = 0$, when $I \neq J$, yielding $\Delta/12$, and $p = 2$ and $q = r = 0$, when $I = J$, resulting in $\Delta/6$. We collect these numbers in a local element matrix:

$$\frac{\Delta}{12}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

The common element matrix entry $\int_{\Omega^{(e)}} \nabla\varphi_I \cdot \nabla\varphi_J \,\mathrm{d}x$, arising from a Laplace term $\nabla^2 u$, can also easily be computed by the formulas above. We have

$$\nabla\varphi_I \cdot \nabla\varphi_J = \frac{\Delta^2}{4}(\beta_I\beta_J + \gamma_I\gamma_J) = \text{const},$$

so that the element matrix entry becomes $\frac{1}{4}\Delta^3(\beta_I\beta_J + \gamma_I\gamma_J)$.

From an implementational point of view, one will work with local vertex numbers $r = 0, 1, 2$, parameterize the coefficients in the basis functions by r, and look up vertex coordinates through $q(e,r)$.

Similar formulas exist for integration of P1 elements in 3D.

7.6 A glimpse of the mathematical theory of the finite element method

Almost all books on the finite element method that introduces the abstract variational problem $a(u,v) = L(v)$ spend considerable pages on deriving error estimates and other properties of the approximate solution. The machinery with function spaces and bilinear and linear forms has the great advantage that a very large class of PDE problems can be analyzed in a unified way. This feature is often taken as an advantage of finite element methods over finite difference and volume methods. Since there are so many excellent textbooks on the mathematical properties of finite element methods [6, 1, 2, 4, 3, 7], this text will not repeat the theory, but give a glimpse of typical assumptions and general results for elliptic PDEs.

Remark. The mathematical theory of finite element methods is primarily developed for to stationary PDE problems of elliptic nature whose solutions are smooth. However, such problems can be solved with the desired accuracy by most numerical methods and pose no difficulties. Time-dependent problems, on the other hand, easily lead to non-physical features in the numerical solutions and therefore requires more care and knowledge by the user. Our focus on the accuracy of the finite element method will of this reason be centered around time-dependent problems, but then we need a different set of tools for the analysis. These tools are based on converting finite element equations to finite difference form and studying Fourier wave components.

Abstract variational forms. To list the main results from the mathematical theory of finite elements, we consider linear PDEs with an abstract variational form

$$a(u,v) = L(v) \quad \forall v \in V\,.$$

This is the discretized problem (as usual in this book) where we seek $u \in V$. The weak formulation of the corresponding continuous problem, fulfilled by the exact solution $u_{\mathrm{e}} \in V_{\mathrm{e}}$ is here written as

$$a(u_{\mathrm{e}},v) = L(v) \quad \forall v \in V_{\mathrm{e}}\,.$$

The space V is finite dimensional (with dimension $N+1$), while V_{e} is infinite dimensional. Normally The hope is that $u \to u_{\mathrm{e}}$ as $N \to \infty$ and $V \to V_{\mathrm{e}}$.

Example on an abstract variational form and associated spaces. Consider the problem $-u''(x) = f(x)$ on $\Omega = [0,1]$, with $u(0) = 0$ and $u'(1) = \beta$. The weak form is

$$a(u,v) = \int_0^1 u'v'dx, \quad L(v) = \int_0^1 fvdx + \beta v(1)\,.$$

The space V for the approximate solution u can be chosen in many ways as previously described. The exact solution u_{e} fulfills $a(u,v) = L(v)$ for all v in V_{e}, and to specify what V_{e} is, we need to introduce *Hilbert spaces*. The Hilbert space $L^2(\Omega)$ consists of all functions that are square-integrable on Ω:

$$L^2(\Omega) = \left\{ \int_\Omega v^2 dx < \infty \right\}\,.$$

The space V_{e} is the space of all functions whose first-order derivative is also square-integrable:

$$V_{\mathrm{e}} = H_0^1(\Omega) = \left\{ v \in L^2(\Omega) \,|\, \frac{dv}{dx} \in L^2(\Omega), \text{ and } v(0) = 0 \right\}\,.$$

The requirements of square-integrable zeroth- and first-order derivatives are motivated from the formula for $a(u,v)$ where products of the first-order derivatives are to be integrated on Ω.

The Sobolev space $H_0^1(\Omega)$ has an inner product

$$(u,v)_{H^1} = \int_\Omega (uv + \frac{du}{dx}\frac{dv}{dx})dx,$$

and associated norm

$$||v||_{H^1} = \sqrt{(v,v)_{H^1}}\,.$$

Assumptions. A set of general results builds on the following assumptions. Let V_{e} be an infinite-dimensional inner-product space such that $u_{\mathrm{e}} \in V_{\mathrm{e}}$. The space has an associated norm $||v||$ (e.g., $||v||_{H^1}$ in the example above with $V_{\mathrm{e}} = H_0^1(\Omega)$).

1. $L(v)$ is linear in its argument.
2. $a(u, v)$ is a bilinear in its arguments.
3. $L(v)$ is bounded (also called continuous) if there exists a positive constant c_0 such that $|L(v)| \leq c_0||v|| \ \forall v \in Vex$.
4. $a(u, v)$ is bounded (or continuous) if there exists a positive constant c_1 such that $|a(u, v)| \leq c_1||u||||v|| \ \forall u, v \in V_{\mathrm{e}}$.
5. $a(u, v)$ is elliptic (or coercive) if there exists a positive constant c_2 such that $a(v, v) \geq c_2||v||^2 \ \forall v \in V_{\mathrm{e}}$.
6. $a(u, v)$ is symmetric: $a(u, v) = a(v, u)$.

Based on the above assumptions, which must be verified in each specific problem, one can derive some general results that are listed below.

Existence and uniqueness. There exists a unique solution of the problem *find* $u_{\mathrm{e}} \in V_{\mathrm{e}}$ such that

$$a(u_{\mathrm{e}}, v) = L(v) \quad \forall v \in V_{\mathrm{e}}\,.$$

(This result is known as the Lax-Milgram Theorem.)

Stability. The solution $u_{\mathrm{e}} \in V_{\mathrm{e}}$ obeys the stability estimate

$$||u|| \leq \frac{c_0}{c_2}\,.$$

Equivalent minimization problem. The solution $u_{\mathrm{e}} \in V_{\mathrm{e}}$ also fulfills the minimization problem

$$\min_{v \in V_{\mathrm{e}}} F(v), \quad F(v) = \frac{1}{2}a(v, v) - L(v)\,.$$

Best approximation principle. The *energy norm* is defined as

$$||v||_a = \sqrt{a(v, v)}\,.$$

The discrete solution $u \in V$ is the best approximation in energy norm,

$$||u_{\mathrm{e}} - u||_a \leq ||u_{\mathrm{e}} - v||_a \quad \forall v \in V\,.$$

This is quite remarkable: once we have V (i.e., a mesh), the Galerkin method finds the best approximation in this space. In the example above, we have $||v||_a = \int_0^1 (v')^2 dx$, so the derivative u' is closer to u_{e}' than any other possible function in V:

$$\int_0^1 (u_{\mathrm{e}}' - u')^2 dx \leq \int_0^1 (u' - v')dx \quad \forall v \in V\,.$$

Best approximation property in the norm of the space. If $||v||$ is the norm associated with V_{e}, we have another best approximation property:

$$||u_{\mathrm{e}} - u|| \leq \left(\frac{c_1}{c_2}\right)^{\frac{1}{2}} ||u_{\mathrm{e}} - v|| \quad \forall v \in \boldsymbol{V}\,.$$

Symmetric, positive definite coefficient matrix. The discrete problem $a(u,v) = L(v)\ \forall v \in V$ leads to a linear system $Ac = b$, where the coefficient matrix A is symmetric ($A^T = A$) and positive definite ($x^T Ax > 0$ for all vectors $x \neq 0$). One can then use solution methods that demand less storage and that are faster and more reliable than solvers for general linear systems. One is also guaranteed the existence and uniqueness of the discrete solution u.

Equivalent matrix minimization problem. The solution c of the linear system $Ac = b$ also solves the minimization problem $\min_w(\frac{1}{2}w^T Aw - b^T w$ in the vector space $\mathbb{R}^{N+1}$.

A priori error estimate for the derivative. In our sample problem, $-u'' = f$ on $\Omega = [0,1]$, $u(0) = 0$, $u'(1) = \beta$, one can derive the following error estimate for Lagrange finite element approximations of degree s:

$$\left(\int_0^1 (u_{\mathrm{e}}' - u')^2 dx\right)^{\frac{1}{2}} \leq Ch^s ||u_{\mathrm{e}}||_{H^{s+1}},$$
$$||u_{\mathrm{e}} - u|| \leq Ch^2 g(u_{\mathrm{e}}),$$

where $||u||_{H^{s+1}}$ is a norm that integrates the sum of the square of all derivatives up to order $s+1$, $g(u_{\mathrm{e}})$ is the integral of $(u'')^2$, C is a constant, and h is the maximum cell length. The estimate shows that choosing elements with higher-degree polynomials (large s) requires more smoothness in u_{e} since higher-order derivatives need to be square-integrable.

A consequence of the error estimate is that $u' \to u_{\mathrm{e}}'$ as $h \to 0$, i.e., the approximate solution converges to the exact one.

The constant C in this and the next estimate depends on the shape of triangles in 2D and tetrahedra in 3D: squeezed elements with a small angle lead to a large C, and such deformed elements are not favorable for the accuracy.

One can generalize the above estimate to the general problem class $a(u,v) = L(v)$: the error in the derivative is proportional to h^s. Note that the expression $||u_{\mathrm{e}} - u||$ in the example is $||u_{\mathrm{e}} - u||_{H^1}$ so it involves the sum of the zeroth and first derivative. The appearance of the derivative makes the error proportional to h^s - if we only look at the solution it converges as h^{s+1} (see below).

The above estimate is called an *a priori* estimate because the bound contains the exact solution, which is not computable. There are also *a posteriori* estimates where the bound involves the approximation u, which is available in computations.

A priori error estimate for the solution. The finite element solution of our sample problem, using P1 elements, fulfills

$$\left(\int_0^1 (u_{\mathrm{e}} - u)^2 dx\right)^{\frac{1}{2}} \leq Ch^2 g(u_{\mathrm{e}})\,.$$

This estimate shows that the error converges as h^2 for P1 elements. An equivalent finite difference method, see Section 3.3, is known to have an error proportional to h^2, so the above estimate is expected. In general, the convergence is h^{s+1} for elements with polynomials of degree s. Note that the estimate for u' is proportional to h raised to one power less.

8 Summary

- When approximating f by $u = \sum_j c_j\varphi_j$, the least squares method and the Galerkin/projection method give the same result. The interpolation/collocation method is simpler and yields different (mostly inferior) results.
- Fourier series expansion can be viewed as a least squares or Galerkin approximation procedure with sine and cosine functions.
- Basis functions should optimally be orthogonal or almost orthogonal, because this gives little round-off errors when solving the linear system, and the coefficient matrix becomes diagonal or sparse.
- Finite element basis functions are *piecewise* polynomials, normally with discontinuous derivatives at the cell boundaries. The basis functions overlap very little, leading to stable numerics and sparse matrices.
- To use the finite element method for differential equations, we use the Galerkin method or the method of weighted residuals to arrive at a variational form. Technically, the differential equation is multiplied by a test function and integrated over the domain. Second-order derivatives are integrated by parts to allow for typical finite element basis functions that have discontinuous derivatives.

- The least squares method is not much used for finite element solution of differential equations of second order, because it then involves second-order derivatives which cause trouble for basis functions with discontinuous derivatives.
- We have worked with two common finite element terminologies and associated data structures (both are much used, especially the first one, while the other is more general):
  1. *elements*, *nodes*, and *mapping between local and global node numbers*
  2. an extended element concept consisting of *cell*, *vertices*, *degrees of freedom*, *local basis functions*, *geometry mapping*, and *mapping between local and global degrees of freedom*
- The meaning of the word "element" is multi-fold: the geometry of a finite element (also known as a cell), the geometry and its basis functions, or all information listed under point 2 above.
- One normally computes integrals in the finite element method element by element (cell by cell), either in a local reference coordinate system or directly in the physical domain.
- The advantage of working in the reference coordinate system is that the mathematical expressions for the basis functions depend on the element type only, not the geometry of that element in the physical domain. The disadvantage is that a mapping must be used, and derivatives must be transformed from reference to physical coordinates.
- Element contributions to the global linear system are collected in an element matrix and vector, which must be assembled into the global system using the degree of freedom mapping (`dof_map`) or the node numbering mapping (`elements`), depending on which terminology that is used.
- Dirichlet conditions, involving prescribed values of u at the boundary, are mathematically taken care of via a boundary function that takes on the right Dirichlet values, while the basis functions vanish at such boundaries. The finite element method features a general expression for the boundary function. In software implementations, it is easier to drop the boundary function and the requirement that the basis functions must vanish on Dirichlet boundaries and instead manipulate the global matrix system (or the element matrix and vector) such that the Dirichlet values are imposed on the unknown parameters.
- Neumann conditions, involving prescribed values of the derivative (or flux) of u, are incorporated in boundary terms arising from integrating terms with second-order derivatives by part. Forgetting to account for the boundary terms implies the condition $\partial u/\partial n = 0$ at parts of the boundary where no Dirichlet condition is set.

9 Exercises

Exercise 1: Refactor functions into a more general class

Section 1.2 lists three functions for computing the analytical solution of some simple model problems. There is quite some repetitive code, suggesting that the functions can benefit from being refactored into a class hierarchy, where the super class solves $-(a(x)u'(x))' = f(x)$ and where subclasses define the equations for the boundary conditions in a model. Make a method for returning the residual in the differential equation and the boundary conditions when the solution is inserted in these equations. Create a test function that verifies that all three residuals vanish for each of the model problems in Section 1.2. Also make a method that returns the solution either as `sympy` expression or as a string in LaTeX format. Add a fourth subclass for the problem $-(au')' = f$ with a Robin boundary condition:

$$u(0) = 0, \quad -u'(L) = C(u - D)\,.$$

Demonstrate the use of this subclass for the case $f = 0$ and $a = \sqrt{1+x}$.

Solution. This is an exercise in software engineering. The model-specific information is related to the boundary conditions only. We can then let the super class take care of the differential equation and the solution process, while subclasses provide a method `get_bc` to return the symbolic expressions for the boundary equations.

The super class may be coded as shown below.

```
import sympy as sym
x, L, C, D, c_0, c_1, = sym.symbols('x L C D c_0 c_1')

class TwoPtBoundaryValueProblem(object):
    """
    Solve -(a*u')' = f(x) with boundary conditions
    specified in subclasses (method get_bc).
    a and f must be sympy expressions of x.
    """
    def __init__(self, f, a=1, L=L, C=C, D=D):
        """Default values for L, C, D are symbols."""
        self.f = f
        self.a = a
        self.L = L
        self.C = C
        self.D = D

        # Integrate twice
        u_x = - sym.integrate(f, (x, 0, x)) + c_0
        u = sym.integrate(u_x/a, (x, 0, x)) + c_1
        # Set up 2 equations from the 2 boundary conditions and solve
        # with respect to the integration constants c_0, c_1
        eq = self.get_bc(u)
        eq = [sym.simplify(eq_) for eq_ in eq]
        print 'BC eq:', eq
        self.u = self.apply_bc(eq, u)
```

```
    def apply_bc(self, eq, u):
        # Solve BC eqs respect to the integration constants
        r = sym.solve(eq, [c_0, c_1])
        # Substitute the integration constants in the solution
        u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
        u = sym.simplify(sym.expand(u))
        return u

    def get_solution(self, latex=False):
        return sym.latex(self.u, mode='plain') if latex else self.u

    def get_residuals(self):
        """Return the residuals in the equation and BCs."""
        R_eq = sym.diff(sym.diff(self.u, x)*self.a, x) + self.f
        R_0, R_L = self.get_bc(self.u)
        residuals = [sym.simplify(R) for R in R_eq, R_0, R_L]
        return residuals

    def get_bc(self, u):
        raise NotImplementedError(
            'class %s has not implemented get_bc' %
            self.__class__.__name__)
```

The various subclasses deal with the boundary conditions of the various model problems:

```
class Model1(TwoPtBoundaryValueProblem):
    """u(0)=0, u(L)=D."""
    def get_bc(self, u):
        return [u.subs(x, 0)-0,               # x=0 condition
                u.subs(x, self.L) - self.D]   # x=L condition

class Model2(TwoPtBoundaryValueProblem):
    """u'(0)=C, u(L)=D."""
    def get_bc(self, u):
        return [sym.diff(u,x).subs(x, 0) - self.C, # x=0 cond.
                u.subs(x, self.L) - self.D]        # x=L cond.

class Model3(TwoPtBoundaryValueProblem):
    """u(0)=C, u(L)=D."""
    def get_bc(self, u):
        return [u.subs(x, 0) - self.C,
                u.subs(x, self.L) - self.D]
```

A suitable test function gets quite compact:

```
def test_TwoPtBoundaryValueProblem():
    f = 2
    model = Model1(f)
    print 'Model 1, u:', model.get_solution()
    for R in model.get_residuals():
        assert R == 0

    f = x
    model = Model2(f)
    print 'Model 2, u:', model.get_solution()
    for R in model.get_residuals():
        assert R == 0

    f = 0
    a = 1 + x**2
```

```
    model = Model3(f, a=a)
    print 'Model 3, u:', model.get_solution()
    for R in model.get_residuals():
        assert R == 0
```

The fourth model is just about defining the boundary conditions as equations:

```
class Model4(TwoPtBoundaryValueProblem):
    """u(0)=0, -u'(L)=C*(u-D)."""
    def get_bc(self, u):
        return [u.subs(x, 0) - 0,
                -sym.diff(u, x).subs(x, self.L) -
                self.C*(u.subs(x, self.L) - self.D)]
```

A demo function goes like

```
def demo_Model4():
    f = 0
    model = Model4(f, a=sym.sqrt(1+x))
    print 'Model 4, u:', model.get_solution()
```

The printout shows that the solution is

$$u(x) = \frac{2CD\sqrt{1+L}(\sqrt{1+x}-1}{2C\sqrt{1+L}+2C+1}.$$

Filename: `uxx_f_sympy_class`.

Exercise 2: Compute the deflection of a cable with sine functions

A hanging cable of length L with significant tension T has a deflection $w(x)$ governed by

$$Tw''(x) = \ell(x),$$

where $\ell(x)$ the vertical load per unit length. The cable is fixed at $x = 0$ and $x = L$ so the boundary conditions become $w(0) = w(L) = 0$. The deflection w is positive upwards, and ℓ is positive when it acts downwards.

If we assume a constant load $\ell(x) = \text{const}$, the solution is expected to be symmetric around $x = L/2$. For a function $w(x)$ that is symmetric around some point x_0, it means that $w(x_0 - h) = w(x_0 + h)$, and then $w'(x_0) = \lim_{h\to 0}(w(x_0 + h) - w(x_0 - h))/(2h) = 0$. We can therefore utilize symmetry to halve the domain. We then seek $w(x)$ in $[0, L/2]$ with boundary conditions $w(0) = 0$ and $w'(L/2) = 0$.

The problem can be scaled by introducing dimensionless independent and dependent variables,

$$\bar{x} = \frac{x}{L/2}, \quad \bar{u} = \frac{w}{w_c},$$

where w_c is a characteristic size of w. Inserted in the problem for w,

$$\frac{4Tw_c}{L^2}\frac{d^2\bar{u}}{d\bar{x}^2} = \ell \ (= \text{const})\,.$$

A desire is to have u and its derivatives about unity, so choosing w_c such that $|d^2\bar{u}/d\bar{x}^2| = 1$ is an idea. Then $w_c = \frac{1}{4}\ell L^2/T$, and the problem for the scaled vertical deflection u becomes

$$u'' = 1, \quad x \in (0,1), \quad u(0) = 0,\ u'(1) = 0\,.$$

Observe that there are no physical parameters in this scaled problem. From now on we have for convenience renamed x to be the scaled quantity $\bar{x}$.

a) Find the exact solution for the deflection u.

Solution. Exercise 1 or Section 1.2 features tools for finding the analytical solution of this differential equation. The present model problem is close to model 2 in Section 1.2. We can modify the `model2` function:

```
def model():
    """Solve u'' = -1, u(0)=0, u'(1)=0."""
    import sympy as sym
    x, c_0, c_1, = sym.symbols('x c_0 c_1')
    u_x = sym.integrate(1, (x, 0, x)) + c_0
    u = sym.integrate(u_x, (x, 0, x)) + c_1
    r = sym.solve([u.subs(x,0) - 0,
                   sym.diff(u,x).subs(x, 1) - 0],
                  [c_0, c_1])
    u = u.subs(c_0, r[c_0]).subs(c_1, r[c_1])
    u = sym.simplify(sym.expand(u))
    return u
```

The solution becomes

$$u(x) = \frac{1}{2}x(x-2)\,.$$

Plotting $u(x)$ shows that $|u| \in [0, \frac{1}{2}]$ which is compatible with the aim of the scaling, i.e., to have u of size *about* unity (at least not very small or very large).

b) A possible function space is spanned by $\psi_i = \sin((2i+1)\pi x/2)$, $i = 0, \ldots, N$. These functions fulfill the necessary condition $\psi_i(0) = 0$, but they also fulfill $\psi_i'(1) = 0$ such that both boundary conditions are fulfilled by the expansion $u = \sum_j c_j\varphi_j$.

Use a Galerkin and a least squares method to find the coefficients c_j in $u(x) = \sum_j c_j\psi_j$. Find how fast the coefficients decrease in magnitude by looking at c_j/c_{j-1}. Find the error in the maximum deflection at $x = 1$ when only one basis function is used ($N = 0$).

Hint. In this case, where the basis functions and their derivatives are orthogonal, it is easiest to set up the calculations by hand and use `sympy` to help out with the integrals.

Solution. With $u = \sum_{j=0}^{N} c_j \psi_j(x)$ the residual becomes

$$R = 1 - u'' = 1 + \sum_{j=0}^{N} c_j \psi_j''(x) = 1 + \sum_{j=0}^{N} c_j (2j+1)^2 \frac{\pi^2}{4} \sin((2j+1)\frac{\pi x}{2}) \,.$$

Least squares method
The minimization of $\int_0^1 R^2 dx$ leads to the equations

$$(R, \frac{\partial R}{\partial c_i}) = 0, \quad i = 0, \ldots, N \,.$$

We find that

$$\frac{\partial R}{\partial c_i} = (2i+1)^2 \frac{\pi^2}{4} \sin((2i+1)\frac{\pi x}{2}),$$

so the governing equations become

$$(1 + \sum_{j=0}^{N} c_j (2j+1)^2 \frac{\pi^2}{4} \sin((2j+1)\frac{\pi x}{2}), (2i+1)^2 \frac{\pi^2}{4} \sin((2i+1)\frac{\pi x}{2}) = 0 \,.$$

By linearity of the inner product (or integral) this expression can be reordered to

$$\sum_{j=0}^{N} c_j ((2j+1)^2 \frac{\pi^2}{4} \sin((2j+1)\frac{\pi x}{2}), (2i+1)^2 \frac{\pi^2}{4} \sin((2i+1)\frac{\pi x}{2}) = \\ - (1, (2i+1)^2 \frac{\pi^2}{4} \sin((2i+1)\frac{\pi x}{2})),$$

which is nothing but a linear system

$$\sum_{j=0}^{N} A_{i,j} c_j = b_i, \quad i = 0, \ldots, N,$$

with

$$A_{i,j} = (2j+1)^4 \frac{\pi^4}{16} \int_0^1 \sin((2j+1)\frac{\pi x}{2}) \sin((2i+1)\frac{\pi x}{2}) dx,$$
$$b_i = -(2i+1)^2 \frac{\pi^2}{4} \int_0^1 \sin((2i+1)\frac{\pi x}{2}) dx$$

Orthogonality of the sine functions $\sin(k\pi x/2)$ on $[0,1]$ for integer k implies that $A_{i,j} = 0$ for $i \neq j$, and $A_{i,i}$ can be computed by `sympy`:

```
>>> from sympy import *
>>> i = symbols('i', integer=True)
>>> x = symbols('x', real=True)
>>> integrate(sin(i*pi*x/2)**2, (x, 0, 1))
1/2
```

Therefore,

$$A_{i,j} = \begin{cases} 0, & i \neq j \\ \frac{1}{2}(2i+1)^4 \frac{\pi^4}{16}, & i = j \end{cases}$$

The right-hand side can also be computed by `sympy`:

```
>>> integrate(sin((2*i+1)*pi*x/2), (x, 0, 1))
2/(pi*(2*i+1))
```

One should always be skeptical to symbolic software and integration of periodic functions like the sine and cosine since the answers can be too simplistic (see subexercise d!). A general test is to perform numerical integration with lots of sampling points to (partially) verify the symbolic formula. Here is an application of the midpoint rule:

```
def midpoint_rule(f, M=100000):
    """Integrate f(x) over [0,1] using M intervals."""
    from numpy import sum, linspace
    dx = 1.0/M                          # interval length
    x = linspace(dx/2, 1-dx/2, M)    # integration points
    return dx*sum(f(x))

def check_integral_b():
    from numpy import pi, sin
    for i in range(12):
        exact = 2/(pi*(2*i+1))
        numerical = midpoint_rule(
            f=lambda x: sin((2*i+1)*pi*x/2))
        print i, abs(exact - numerical)
```

The output shows that the difference between numerical and exact integration is about 10^{-11}, which is "small" (and gets smaller by just increasing `M`). This result brings evidence that the `sympy` answer is correct. Alternatively, in this simple case, we can easily calculate the anti-derivative. It goes like

$$-\frac{2}{\pi(2i+1)}\cos((2k+1)\frac{\pi x}{2}),$$

and for $x = 1$ we get $\cos\frac{\pi}{2}$, $\cos 3\frac{\pi}{2}$, $\cos 5\frac{\pi}{2}$, and so on, which all evaluates to zero, and since the cosine is 1 for $x = 0$, the formula found by `sympy` is correct.

We then get

$$b_i = -(2i+1)^2\frac{\pi^2}{4}\frac{2}{\pi(2i+1)} = -\frac{1}{2}(2i+1)\pi,$$

and consequently,

$$c_i = \frac{b_i}{A_{i,i}} = -\frac{\frac{1}{2}(2i+1)\pi}{\frac{1}{2}(2i+1)^4}\frac{\pi^4}{16} = -\frac{16}{\pi^3(2i+1)^3}\,.$$

Galerkin's method

The Galerkin method applied to this problem starts with

$$(u'', v) = (1, v) \quad \forall v \in V,$$

and the requirement that $v(0) = 0$ since $u(0) = 0$. Integration by parts and using $u'(1) = 0$ and $v(0) = 0$ makes the boundary term vanish, and the variational form becomes

$$(u', v') = -(1, v) \quad \forall v \in V\,.$$

Inserting $u = \sum_{j=0}^{N} c_j\psi_j(x)$ and $v = \psi_i$ leads to

$$\sum_{j=0}^{N}(\psi_j', \psi_i')c_j = (1, \psi_i), \quad i = 0, \ldots, N\,.$$

With $\psi_i = \sin((2i+1)\frac{\pi x}{2})$ the matrix entries become

$$A_{i,j} = (2i+1)(2j+1)\frac{\pi^2}{4}\int_0^1 \cos((2i+1)\frac{\pi x}{2})\cos((2j+1)\frac{\pi x}{2})dx\,.$$

Orthogonality of the cosine functions implies $A_{i,j} = 0$ for $i \neq j$, and $A_{i,i}$ is computed by integrating the square of the cosine function,

```
>>> integrate(cos((k+1)*pi*x/2)**2, (x, 0, 1))
1/2
```

Now,

$$A_{i,i} = (2i+1)^2\frac{\pi^2}{4}\frac{1}{2} = \frac{1}{8}(2i+1)^2\pi^2\,.$$

The right-hand side has almost the same integral as in the least squares case,

$$b_i = -\int_0^1 \sin((2i+1)\frac{\pi x}{2})dx = -\frac{2}{\pi(2i+1)}\,.$$

Consequently,

$$c_i = \frac{b_i}{A_{i,i}} = -\frac{16}{\pi^3(2i+1)^3},$$

which is the same result as we obtained in the least squares method.

Decay of coefficients

The coefficients decay,

$$\frac{c_i}{c_{i+1}} = \left(\frac{2i+3}{2i+1}\right)^3 > 0\,.$$

The decay is most pronounced for the first terms:

```
>>> for i in range(10):
...   print (float(2*i+3)/(2*i+1))**3
...
27.0
4.62962962963
2.744
2.12536443149
1.82578875171
1.65063861758
1.53618570778
1.4557037037
1.39609200081
1.35019682169
```

Error in one-term solution
Keeping just one term ($N = 0$) means that

$$u(x) = -\frac{16}{\pi^3}\sin(\frac{\pi x}{2})\,.$$

The maximum deflection at $x = 1$ becomes $-16\pi^{-3} = -0.5160$, to be compared with the exact value $-\frac{1}{2}$. The error is 3.2 percent.

c) Visualize the solutions in b) for $N = 0, 1, 20$.

Solution. First we need a function to compute the approximate u in this case:

```
def sine_sum(x, N):
    s = 0
    from numpy import pi, sin, zeros
    u = [] # u[k] is the sum i=0,...,k
    k = 0
    for i in range(N+1):
        s += - 16.0/((2*i+1)**3*pi**3)*sin((2*i+1)*pi*x/2)
        u.append(s.copy())  # important with copy!
    return u
```

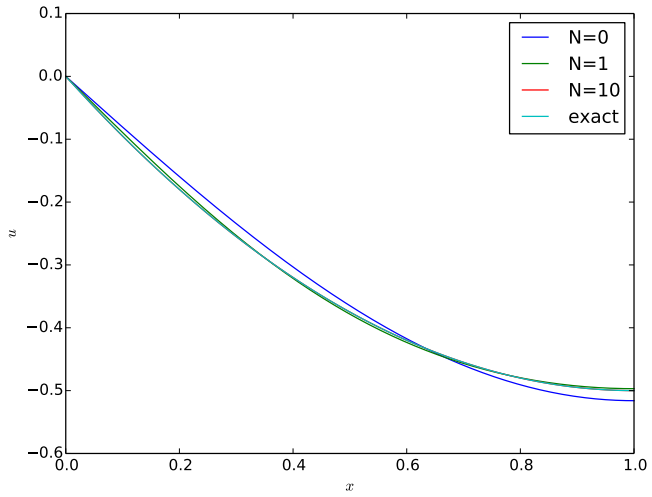
Note the need to append `s.copy()`: doing just `u.append(s)` will make, e.g., `u[0]` a reference to `s`, which at the end of the loop is an array corresponding to the maximum i value.

We also need a function that can create an appropriate plot:

```
def plot_sine_sum():
    from numpy import linspace
    x = linspace(0, 1, 501)  # coordinates for plot
    u = sine_sum(x, N=10)
    u_e = 0.5*x*(x-2)
    N_values = 0, 1, 10
    for k in N_values:
        plt.plot(x, u[k])
```

```
        plt.hold('on')
    plt.plot(x, u_e)
    plt.legend(['N=%d' % k for k in N_values] + ['exact'],
               loc='upper right')
    plt.xlabel('$x$');  plt.ylabel('$u$')
    plt.savefig('tmpc.png'); plt.savefig('tmpc.pdf')
```

The plot shows that the solution for $N = 0$ has a slight deviation from the exact curve, but even $N = 1$ catches up visually with the exact solution (!).



d) The functions in b) were selected such that they fulfill the condition $\psi'(1) = 0$. However, in the Galerkin method, where we integrate by parts, the condition $u'(1) = 0$ is incorporated in the variational form. This leads to the idea of just choosing a simpler basis, namely "all" sine functions $\psi_i = \sin((i+1)\frac{\pi x}{2})$. Will the method adjust the coefficient such that the additional functions compared with those in b) get vanishing coefficients? Or will the additional basis functions improve the solution? Use Galerkin's method.

Solution. According to the calculations in b), the Galerkin method, with $\psi_i = \sin((i+1)\frac{\pi x}{2})$, leads to the almost the same matrix entries on the diagonal:

$$
\begin{aligned}
A_{i,i} &= (i+1)(j+1)\frac{\pi^2}{4}\int_0^1 \cos((i+1)\frac{\pi x}{2})\cos((j+1)\frac{\pi x}{2})dx \\
&= (i+1)^2\frac{\pi^2}{4}\frac{1}{2} = \frac{1}{8}(i+1)^2\pi^2\,.
\end{aligned}
$$

The right-hand side becomes (as before)

$$
b_i = -\int_0^1 \sin((i+1)\frac{\pi x}{2})dx = -\frac{2}{\pi(i+1)}\,.
$$

We may use `sympy` to integrate,

```
>>> integrate(sin((i+1)*pi*x/2), (x, 0, 1))
2/(pi*(i+1))
```

As noted in b), let us be a bit skeptical to this answer and check it. A quick check with numerical integration,

```
def check_integral_d_sympy_answer():
    from numpy import pi, sin
    for i in range(12):
        exact = 2/(pi*(i+1))
        numerical = midpoint_rule(
            f=lambda x: sin((i+1)*pi*x/2))
        print i, abs(exact - numerical)
```

gives the output

```
0 6.54487575247e-12
1 0.31830988621
2 1.96350713466e-11
3 0.159154943092
4 3.27249061183e-11
5 0.106103295473
6 4.58150045679e-11
7 0.0795774715459
8 5.89047144395e-11
9 0.0636619773677
10 7.19949447281e-11
11 0.0530516476973
```

It is clear that for i odd, there are significant differences between the `sympy` answer and the midpoint rule with high resolution!

We therefore need to do hand calculations to investigate this problem further. The anti-derivative is very easy to realize in this case:

$$\begin{aligned}\int_0^1 \sin((i+1)\pi x/2)dx &= -\frac{2}{\pi(i+1)}(\cos((i+1)\frac{\pi}{2}) - \cos(0)) \\ &= \frac{2}{\pi(i+1)}(1 - \cos((i+1)\frac{\pi}{2}))\,.\end{aligned}$$

The value of the cosine expression depends on i, and the first values are

$i=0$	$i=1$	$i=2$	$i=3$
0	-1	0	1

This pattern repeats and is the same for four consecutive values of i. Hence, the integral is $2/(\pi(i+1))$ for even i ($i = 2k$ for integer k, or equivalently: when $i \bmod 2 = 0$). For $i = 4k+1$, or equivalently: when $(i-1) \bmod 4 = 0$, the integral is $4/(\pi(4k+1))$, while for $i = 4k+3$, the integral vanishes. This is a more complicated answer than what `sympy` provides!

We can check our new answers against numerical integration:

```
def check_integral_d():
    from numpy import pi, sin
    for i in range(24):
        if i % 2 == 0:
            exact = 2/(pi*(i+1))
        elif (i-1) % 4 == 0:
            exact = 2*2/(pi*(i+1))
        else:
            exact = 0
        numerical = midpoint_rule(
            f=lambda x: sin((i+1)*pi*x/2))
        print i, abs(exact - numerical)
```

The output now is around 10^{-10} and we take that as a sign that our exact results are reliable.

Carefully check symbolic computations!

The example above shows how `sympy` can fail. Wolfram Alpha[a] does a better job: writing `integrate sin(k*x*pi/2) from 0 to 1` (use `k` instead of `i` since the latter is the imaginary unit) returns the result[b] $4\sin^2(\pi k/4)/(\pi k)$, which coincides with out result.

There are three general techniques to verify a symbolic computation:

- Use alternative software like Wolfram Alpha for comparison
- Check that the result satisfies the problem to be solved
- Make a high-resolution numerical approximation and compare

(The second technique is not so applicable here, since we work with a definite integral, but one could compute the indefinite integral instead, which is done correctly by `sympy`, and discuss values for $x = 1$.)

[a] `http://wolframalpha.com`
[b] `http://www.wolframalpha.com/input/?i=integrate+sin%28k*x*pi%2F2%29+from+0+to+1`

The final result for c_i is now

$$c_i = \frac{b_i}{A_{i,i}} = \begin{cases} -\frac{16}{\pi^3(i+1)^3}, & i \text{ even, or } i \bmod 2 = 0 \\ -\frac{32}{\pi^3(i+1)^3}, & (i-1) \bmod 4 = 0, \\ 0, & (i+1) \bmod 4 = 0 \end{cases}$$

We recognize that for i even, say $i = 2k$ for integer k, we have exactly the same result as in b):

$$-\sum_k \frac{16}{\pi^3(2k+1)^3}\sin((2k+1)x\frac{\pi x}{2}),$$

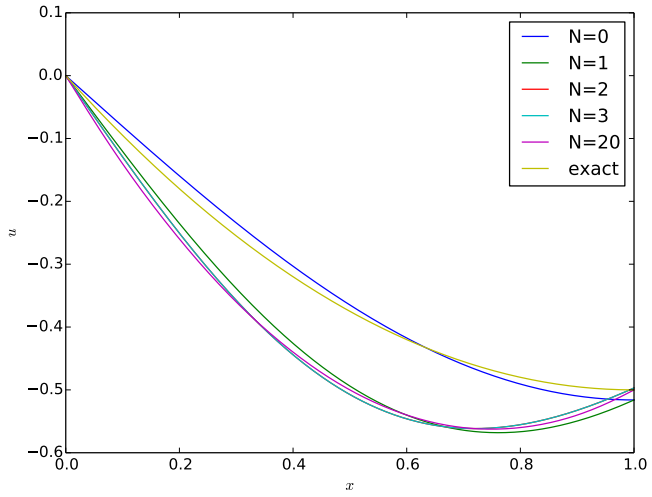
but we get an additional set of terms for $i = 4k + 1$,

$$-\sum_k \frac{32}{\pi^3(i+1)^3}\sin((4k+1)x\frac{\pi x}{2})\,. \tag{95}$$

We can modify the software from c) to compute the approximate u with the present set of basis functions and coefficients:

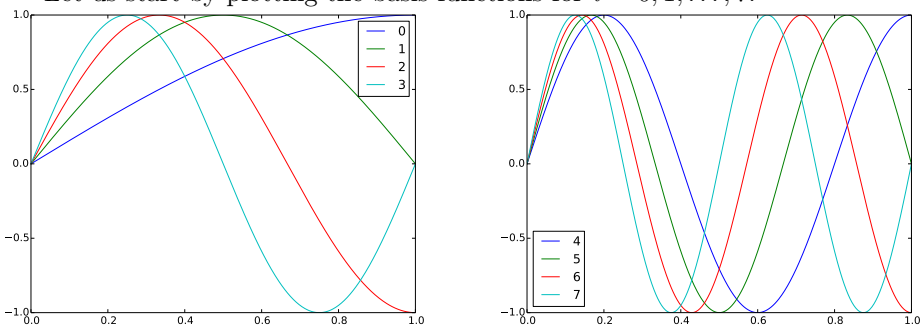
```
def sine_sum_d(x, N):
    s = 0
    from numpy import pi, sin, zeros
    u = []  # u[k] is the sum i=0,...,k
    k = 0
    for i in range(N+1):
        if i % 2 == 0:       # even i
            s +=   - 16.0/((i+1)**3*pi**3)*sin((i+1)*pi*x/2)
        elif (i-1) % 4 == 0:   # 1, 5, 9, 13, 17
            s += - 2*16.0/((i+1)**3*pi**3)*sin((i+1)*pi*x/2)
        else:
            s += 0
        u.append(s.copy())
    return u

def plot_sine_sum_d():
    from numpy import linspace
    x = linspace(0, 1, 501)  # coordinates for plot
    u = sine_sum_d(x, N=20)
    u_e = 0.5*x*(x-2)
    N_values = 0, 1, 2, 3, 20
    for k in N_values:
        plt.plot(x, u[k])
        plt.hold('on')
    plt.plot(x, u_e)
    plt.legend(['N=%d' % k for k in N_values] + ['exact'],
               loc='upper right')
    plt.xlabel('$x$');  plt.ylabel('$u$')
    #plt.axis([0.9, 1, -0.52, -0.49])
    plt.savefig('tmpd.png'); plt.savefig('tmpd.pdf')
```

The approximations for $N = 0, 1, 3, 20$ appear below.



While the approximation for $N = 0$ coincides with the one in b), we see that $N = 1$ and higher values of N lead to a clearly wrong curve. This strange feature has to be investigated!

Let us start by plotting the basis functions for $i = 0, 1, \ldots, 7$:



We observe from the figure that all the basis functions corresponding to even i are symmetric around $x = 1$, which is an important property of the solution. The functions for odd i are anti-symmetric. However, for $i = 3, 7, 11, \ldots$ the basis function has an integer number of periods on $[0, 1]$ so the integral becomes zero, $c_i = 0$, and consequently there is no effect from these functions. The functions corresponding to $i = 1, 5, 9, 13, \ldots$ are anti-symmetric around $x = 1$ with nonzero coefficients. The derivative of an anti-symmetric function at the point of anti-symmetry is unity in size. Since the derivatives of all the basis functions corresponding to even i vanish at $x = 1$, the extra terms ($i = 1, 5, 9, 13, \ldots$) in (95) have a nonzero derivative, resulting in $u'(1) \neq 0$. That is, these terms destroy the solution!

But we computed c_i by a Galerkin method, which is equivalent to a least squares method, which gives us the "best" approximation possible? That is true, but it is the best approximation in the chosen space V. The problem is that we

have populated (or rather polluted) the space V with some basis functions that have a wrong mathematical property: they are anti-symmetric around $x = 1$.

e) Now we drop the symmetry condition at $x = 1$ and extend the domain to $[0, 2]$ such that it covers the entire (scaled) physical cable. The problem now reads

$$u'' = 1, \quad x \in (0, 2), \quad u(0) = u(2) = 0\,.$$

This time we need basis functions that are zero at $x = 0$ and $x = 2$. The set $\sin((i+1)\frac{\pi x}{2})$ from d) is a candidate since they vanish $x = 2$ for any i. Compute the approximation in this case. Why is this approximation without the problem that this set of basis functions introduced in d)?

Solution. The formulas are almost the same as in d), only the integration domain is different. Since the sine functions or orthogonal on $[0, 1]$, they are also orthogonal on $[0, 2]$. Because

```
>>> integrate(cos((i+1)*pi*x/2)**2, (x, 0, 2))
1
```

we get (in Galerkin's method)

$$\begin{aligned} A_{i,i} &= (i+1)(j+1)\frac{\pi^2}{4}\int_0^2 \cos((i+1)\frac{\pi x}{2})\cos((i+1)\frac{\pi x}{2})dx \\ &= (i+1)^2\frac{\pi^2}{4}\,. \end{aligned}$$

and

$$b_i = -\int_0^2 \sin((i+1)\frac{\pi x}{2})dx = \frac{2}{\pi(i+1)}(\cos((i+1)\pi) - 1)\,.$$

We have that $\cos((i+1)\pi = -1$ for i even and $\cos((i+1)\pi = 1$ for i odd. That is,

$$b_i = \begin{cases} -\frac{4}{\pi(i+1)}, & i \text{ even} \\ 0, & i \text{ odd} \end{cases}$$

The coefficients become

$$c_i = \frac{b_i}{A_{i,i}} = \begin{cases} -\frac{16}{\pi^3(i+1)^3}, & i \text{ even} \\ 0, & i \text{ odd} \end{cases}$$

Introducing $i = 2k$ and then switching from k to i as summation index gives $c_i = -\frac{16}{\pi^3(2i+1)^3}$ and

$$u(x) = -\sum_{i=0}^{N} \frac{16}{\pi^3(2i+1)^3}\sin((i+1)\frac{\pi x}{2}),$$

which is the same expansion as in b).

The reason why the basis functions $\psi_i = \sin((i+1)\frac{\pi x}{2})$ work well in this case is that the problematic functions for $i = 1, 5, \ldots$ in d) now live on $[0, 2]$ instead of $[0, 1]$. On $[0, 2]$ these functions have an integer number of periods such that the integral from 0 to 2 becomes zero. These basis functions are therefore excluded from the expansion since their coefficients vanish. The lesson learned is that two equivalent boundary value problems may make different demands to the basis functions.
Filename: `cable_sin`.

Exercise 3: Compute the deflection of a cable with power functions

a) Repeat Exercise 2 b), but work with the space

$$V = \text{span}\{x, x^2, x^3, x^4, \ldots\}\,.$$

Choose the dimension of V to be 4 and observe that the exact solution is recovered by the Galerkin method.

Hint. Use the `solver` function from `varform1D.py`.

Solution. The Galerkin formulation of $u'' = 1$, $u(0) = 0$, $u'(1) = 0$, reads

$$(u', v') = -(1, v) \quad \forall v \in V,$$

and the linear system becomes

$$\sum_{j=}^{N} (\psi_i', \psi_j') c_j = -(1, \psi_i), \quad i = 0, 1, \ldots, N\,.$$

The `varform1D.solver` function needs a function specifying the integrands on the left- and right-hand sides of the variational formulation. Moreover, we must compute a dictionary of ψ_i and ψ_i'. The appropriate code becomes

```
from varform1D import solver
import sympy as sym
x, b = sym.symbols('x b')
f = 1

# Compute basis functions and their derivatives
N = 4
psi = {0: [x**(i+1) for i in range(N+1)]}
psi[1] = [sym.diff(psi_i, x) for psi_i in psi[0]]

# Galerkin

def integrand_lhs(psi, i, j):
    return psi[1][i]*psi[1][j]

def integrand_rhs(psi, i):
```

```
    return -f*psi[0][i]

Omega = [0, 1]

u, c = solver(integrand_lhs, integrand_rhs, psi, Omega,
              verbose=True, symbolic=True)
print 'Galerkin solution u:', sym.simplify(sym.expand(u))
```

Running this code gives the output

```
solution u: x*(x - 2)/2
```

which coincides with the exact solution ($c_3 = c_4 = 0$).

b) What happens if we use a least squares method for this problem with the basis in a)?

Solution. The least squares formulation leads to

$$(R, \frac{\partial R}{\partial c_i}) = 0, \quad i = 0, \ldots, N,$$

with

$$R = 1 - u'' = 1 - \sum_j c_j \psi_j'' \,.$$

We have

$$\frac{\partial R}{\partial c_i} = \psi_i'',$$

leading to the equations

$$(1 + \sum_j c_j \psi_j'', \psi_i''), \quad i = 0, \ldots, N,$$

which is a linear system

$$\sum_{j=0}^{N} (\psi_j'', \psi_i'') = (-1, \psi_i''), \quad i = 0, \ldots, N \,.$$

The fundamental problem with the basis in a) is that $\psi_0'' = 0$, so if power functions of x are wanted, we need to work with the basis $V = \text{span}\{x^2, x^3, \ldots\}$. If we do so, we can easily modify the code from a),

```
# Least squares
psi = {0: [x**(i+2) for i in range(N+1)]}
psi[1] = [sym.diff(psi_i, x) for psi_i in psi[0]]
psi[2] = [sym.diff(psi_i, x) for psi_i in psi[1]]

def integrand_lhs(psi, i, j):
    return psi[2][i]*psi[2][j]
```

```
def integrand_rhs(psi, i):
    return -f*psi[2][i]

Omega = [0, 1]

u, c = solver(integrand_lhs, integrand_rhs, psi, Omega,
              verbose=True, symbolic=True)
print 'solution u:', sym.simplify(sym.expand(u))
```

The result is $u = -\frac{1}{2}x^2$. This function does not obey $u'(1) = 0$ and is completely wrong. In this least squares method we cannot access the basis function x, which is needed in the exact solution, and we have no means to obtain $u'(1) = 0$.

Remark. There is a modification of the least squares method that can be applied here. The problem $u'' = 1$ must be rewritten as a system of two equations, $u_1' = u_2$, $u_2' = 1$. We expand $u_1 = \sum_{j=0}^{N} c_j\psi_j$ and $u_2 = \sum_{j=0}^{N} d_j\psi_j$. The residuals in both equations are added, squared, and differentiated with respect to c_i and d_i, $i = 0, \ldots, N$. The result is a coupled equation system for the c_i and d_i coefficients.
Filename: `cable_xn`.

Exercise 4: Check integration by parts

Consider the Galerkin method for the problem involving u in Exercise 2. Show that the formulas for c_j are independent of whether we perform integration by parts or not.

Solution. The Galerkin method is

$$(u'', v) = (1, v) \quad \forall v \in V,$$

and with the choice of V we get

$$A_{i,j} = -(i+1)^2\pi^2 \int_0^1 \sin^2((i+1)\frac{\pi x}{2})dx,$$
$$b_i = \int_0^1 \sin((i+1)\frac{\pi x}{2})dx$$

From Exercise 2 we realize that the integrals are the same as in the least squares method, and those results were identical to those of the Galerkin method with integration by parts.
Filename: `cable_integr_by_parts`.

Exercise 5: Compute the deflection of a cable with 2 P1 elements

Solve the problem for u in Exercise 2 using two P1 linear elements. Incorporate the condition $u(0) = 0$ by two methods: 1) excluding the unknown at $x = 0$, 2) keeping the unknown at $x = 0$, but modifying the linear system.

Solution. From Exercise 2, the Galerkin method, after integration by parts, reads

$$(u', v') = -(1, v) \quad \forall v \in V\,.$$

We have two elements, $\Omega^{(0)} = [0, \frac{1}{2}]$ and $\Omega^{(1)} = [\frac{1}{2}, 1]$.

Method 1: Excluding the unknown at $x = 0$

Since $u(0) = 0$, we exclude the value at $x = 0$ as degree of freedom in the linear system. (There is no need for any boundary function.) The expansion reads $u = c_0\varphi_1(x) + c_1\varphi_2(x)$. The element matrix has then only one entry in the first element,

$$\tilde{A}^{(0)} = \frac{1}{h}(1)\,.$$

From element 1 we get the usual element matrix

$$\tilde{A}^{(1)} = \frac{1}{h}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}\,.$$

The element vector in element 0 becomes

$$\tilde{b}^{(0)} = \frac{h}{2}(-1),$$

while the second element gives a contribution

$$\tilde{b}^{(1)} = \frac{h}{2}\begin{pmatrix} -1 \\ -1 \end{pmatrix}\,.$$

Assembling the contributions gives

$$\frac{1}{h}\begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = -\frac{h}{2}\begin{pmatrix} 2 \\ 1 \end{pmatrix}\,.$$

Note that $h = \frac{1}{2}$. Solving this system yields

$$c_0 = -\frac{3}{8}, \quad c_1 = -\frac{1}{2} \quad \Rightarrow \quad u = -\frac{3}{8}\varphi_1(x) - \frac{1}{2}\psi_2(x)\,.$$

Evaluating the exact solution for $x = \frac{1}{2}$ and $x = 1$, we get 3/8 and 1/2, respectively, a result which shows that the numerical solution with P1 is exact at the three node points. The difference between the numerical and exact solution is that the numerical solution varies linearly over the two elements while the exact solution is quadratic.

Method 2: Modifying the linear system

Now we let c_i correspond to the value at node x_i, i.e., all known Dirichlet values become part of the linear system. The expansion is now simply $u = \sum_{i=0}^{2} c_i\varphi_i(x)$, with three unknowns c_0, c_1, and c_2. Now the element matrix in the first and second element are equal. The same is true for the element vectors. Assembling yields

$$\frac{1}{h}\begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = -\frac{h}{2}\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}.$$

The next step is to modify the linear system to implement the Dirichlet condition $c_0 = 0$. We first multiply by $h = \frac{1}{2}$ and replace the first equation by $c_0 = 0$:

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = -\begin{pmatrix} 0 \\ \frac{1}{4} \\ \frac{1}{8} \end{pmatrix}.$$

We see that the remaining 2×2 system is identical to the one previously solved, and the solution is the same.

$$u = 0\varphi_0(x) - \frac{3}{8}\varphi_1(x) - \frac{1}{2}\psi_2(x)\,.$$

Filename: `cable_2P1`.

Exercise 6: Compute the deflection of a cable with 1 P2 element

Solve the problem for u in Exercise 2 using one P2 element with quadratic basis functions.

Solution. The P2 basis functions on a reference element $[-1, 1]$ are

$$\tilde{\varphi}_0(X) = \frac{1}{2}(X-1)X$$
$$\tilde{\varphi}_1(X) = 1 - X^2$$
$$\tilde{\varphi}_2(X) = \frac{1}{2}(X+1)X$$

The element matrix and vector are easily calculated by some lines with `sympy`:

```
import sympy as sym
X, h = sym.symbols('X h')
half = sym.Rational(1, 2)
psi = [half*(X-1)*X, 1-X**2, half*(X+1)*X]
dpsi_dX = [sym.diff(psi[r], X) for r in range(len(psi))]

# Element matrix
# (2/h)*dpsi_dX[r]*(2/h)*dpsi_dX[s]*h/2
```

```
import numpy as np
d = 2
# Use a numpy matrix with general objects to hold A
A = np.empty((d+1, d+1), dtype=object)
for r in range(d+1):
    for s in range(d+1):
        integrand = dpsi_dX[r]*dpsi_dX[s]*2/h
        A[r,s] = sym.integrate(integrand, (X, -1, 1))
print A

# Element vector
# f*psi[r]*h/2, f=1
d = 2
b = np.empty(d+1, dtype=object)
for r in range(d+1):
    integrand = -psi[r]*h/2
    b[r] = sym.integrate(integrand, (X, -1, 1))
print b
```

The formatted element matrix and vector output becomes

```
[[7/(3*h) -8/(3*h) 1/(3*h)]
 [-8/(3*h) 16/(3*h) -8/(3*h)]
 [1/(3*h) -8/(3*h) 7/(3*h)]]
[-h/6 -2*h/3 -h/6]
```

or in mathematical notation:

$$\tilde{A}^{(e)} = \frac{1}{3h}\begin{pmatrix} 7 & -8 & 1 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{pmatrix}, \quad \tilde{b}^{(e)} = -\frac{h}{6}\begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix}.$$

Method 1: Excluding the unknown at $x = 0$

The expansion is $u = c_0\varphi_1(x) + c_1\varphi_2(x)$. The element matrix corresponding to the first element excludes contributions associated with the unknown at the left node, i.e., we exclude row and column 0. In the element vector, we exclude the first entry.

$$\tilde{A}^{(0)} = \frac{1}{3h}\begin{pmatrix} 16 & -8 \\ -8 & 7 \end{pmatrix}, \quad \tilde{b}^{(e)} = -\frac{h}{6}\begin{pmatrix} 4 \\ 1 \end{pmatrix}.$$

Now, $h = 1$. The solution of the linear system

$$\frac{1}{3h}\begin{pmatrix} 16 & -8 \\ -8 & 7 \end{pmatrix}\begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = -\frac{h}{6}\begin{pmatrix} 4 \\ 1 \end{pmatrix}$$

is $c_1 = 3/8$ and $c_2 = 1/2$. As for P1 elements in Exercise 5, the values at the nodes are exact, but this time the variation between the nodes is quadratic, i.e., exact. One P2 element produces the complete, exact solution.

Method 2: Modifying the linear system

This time the expansion reads $u = \sum_{i=0}^{2} c_i\varphi_i(x)$ with three unknowns c_0, c_1, and c_2. The linear system consists of the complete 3×3 element matrix and the corresponding element vector:

$$\frac{1}{3h}\begin{pmatrix} 7 & -8 & 1 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{pmatrix}\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = -\frac{h}{6}\begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix}.$$

The boundary condition is incorporated by replacing the first equation by $c_0 = 0$, but prior to taking that action, we multiply by $3h$ and insert $h = 1$.

$$\begin{pmatrix} 1 & 0 & 0 \\ -8 & 16 & -8 \\ 1 & -8 & 7 \end{pmatrix}\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 0 \\ -2 \\ -\frac{1}{2} \end{pmatrix}.$$

Realizing that $c_0 = 0$, which means we can remove the first column of the system, shows that the equations are the same as above and hence that the solution is identical.
Filename: `cable_1P2`.

Exercise 7: Compute the deflection of a cable with a step load

We consider the deflection of a tension cable as described in Exercise 2: $w'' = \ell$, $w(0) = w(L) = 0$. Now the load is discontinuous:

$$\ell(x) = \begin{cases} \ell_1, & x < L/2, \\ \ell_2, & x \geq L/2 \end{cases} \qquad x \in [0, L]\,.$$

This load is not symmetric with respect to the midpoint $x = L/2$ so the solution loses its symmetry. Scaling the problem by introducing

$$\bar{x} = \frac{x}{L/2}, \quad u = \frac{w}{w_c}, \quad \bar{\ell} = \frac{\ell - \ell_1}{\ell_2 - \ell_1}\,.$$

This leads to a scaled problem on $[0, 2]$ (we rename $\bar{x}$ as x for convenience):

$$u'' = \bar{\ell}(x) = \begin{cases} 1, & x < 1, \\ 0, & x \geq 1 \end{cases} \qquad x \in (0, 1), \quad u(0) = 0,\ u(2) = 0\,.$$

a) Find the analytical solution of the problem.

Hint. Integrate the equation separately for $x < 1$ and $x > 1$. Use the conditions that u and u' must be continuous at $x = 1$.

Solution. For $x < 1$ we get $u_1(x) = C_1 x + C_2$, and the boundary condition $u_1(0) = 0$ implies $C_2 = 0$. For $x > 1$ we get $u_2(x) = \frac{1}{2}x^2 + C_3 x + C_4$. Continuity of $u'(1)$ leads to

$$C_1 = 1 + C_3,$$

and continuity of $u(1)$ means

$$C_1 = \frac{1}{2} + C_3 + C_4,$$

while the condition $u_2(2) = 0$ gives the third equation we need:

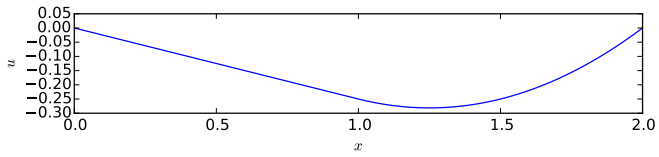
$$2 + 2C_3 + C_4 = 0\,.$$

We use `sympy` to solve them:

```
>>> from sympy import symbols, Rational, solve
>>> C1, C3, C4 = symbols('C1 C3 C4')
>>> solve([C1 - 1 - C3,
           C1 - Rational(1,2) - C3 - C4,
           2 + 2*C3 + C4], [C1,C3,C4])
{C1: -1/4, C4: 1/2, C3: -5/4}
```

Then

$$u(x) = \begin{cases} -\frac{1}{4}x, & x \leq 1, \\ \frac{1}{2}x^2 - \frac{5}{4}x + \frac{1}{2}, & x \geq 1 \end{cases}$$



b) Use $\psi_i = \sin((i+1)\frac{\pi x}{2})$, $i = 0, \ldots, N$ and the Galerkin method to find an approximate solution $u = \sum_j c_j \psi_j$. Plot how fast the coefficients c_j tend to zero (on a log scale).

Solution. The Galerkin formulation of the problem becomes

$$(u', v') = -(\bar{\ell}, v) = \begin{cases} 0, & x \leq 1, \\ -(1, v), & x \geq 1 \end{cases} \qquad \forall v \in V\,.$$

A requirement is that $v(0) = v(2) = 0$ because of the boundary conditions on u. The chosen basis functions fulfill this requirement for any integer i. Inserting

$u = \sum_{j=0}^{N} c_j\psi_j$ and $v = \psi_i$, $i = 0, \ldots, N$, gives as usual the linear system $\sum_j A_{i,j}c_j = b_i$, $i = 0, \ldots, N$, where

$$A_{i,j} = (i+1)(j+1)\frac{\pi^2}{4}\int_0^2 \cos((i+1)\frac{\pi x}{2})\cos((j+1)\frac{\pi x}{2})dx\,.$$

The cosine functions are orthogonal on $[0, 2]$ so $A_{i,j} = 0$ for $i \neq j$, while $A_{i,i}$ is computed (e.g., by `sympy`) as in Exercise 2, part e. The result is

$$A_{i,i} = (i+1)^2\frac{\pi^2}{4}\,.$$

The right-hand side is

$$b_i = -\int_1^2 \sin((i+1)\frac{\pi x}{2})dx = \frac{2}{\pi(i+1)}(\cos((i+1)\pi) - \cos((i+1)\pi/2))\,.$$

(Trying to do the integral in `sympy` gives a complicated expression that needs discussion - it is easier to do all calculations by hand.) We have that $\cos((i+1)\pi = -1$ for i even and $\cos((i+1)\pi = 1$ for i odd, while $\cos((i+1)\pi/2)$ is discussed in Exercise 2, part d. The values of $\cos((i+1)\pi) - \cos((i+1)\pi/2)$ can be summarized in the following table:

$i \bmod 4 = 0$	$(i-1) \bmod 4 = 0$	$(i-2) \bmod 4 = 0$	$(i-3) \bmod 4 = 0$
$-1-0$	$1-(-1)$	$-1-0$	$1-1$

The following function computes the approximate solution:

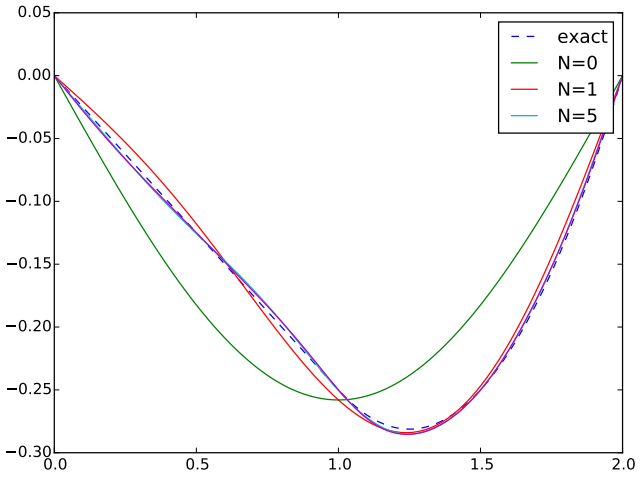
```
def sine_solution(x, N):
    from numpy import pi, sin
    s = 0
    u = [] # u[i] is the solution for N=i
    for i in range(N+1):
        if i % 4 == 0:
            cos_min_cos = -1
        elif (i-1) % 4 == 0:
            cos_min_cos = 2
        elif (i-2) % 4 == 0:
            cos_min_cos = -1
        elif (i-1) % 4 == 0:
            cos_min_cos = 0

        b_i = 2/(pi*(i+1))*cos_min_cos
        A_ii = (i+1)**2*pi**2/4
        c_i = b_i/A_ii
        s += c_i*sin((i+1)*x*pi/2)
        u.append(s.copy())
    return u
```

The exact solution is a function defined in a piecewise way. Below we make an implementation that works both for array and scalar arguments:

```
def exact_solution(x):
    if isinstance(x, np.ndarray):
        return np.where(x < 1, -1./4*x, 0.5*x**2 - 5./4*x + 0.5)
    else:
        return -1./4*x if x < 1 else 0.5*x**2 - 5./4*x + 0.5
```

Now we can make a plot of the exact solution and approximate solutions for various N:

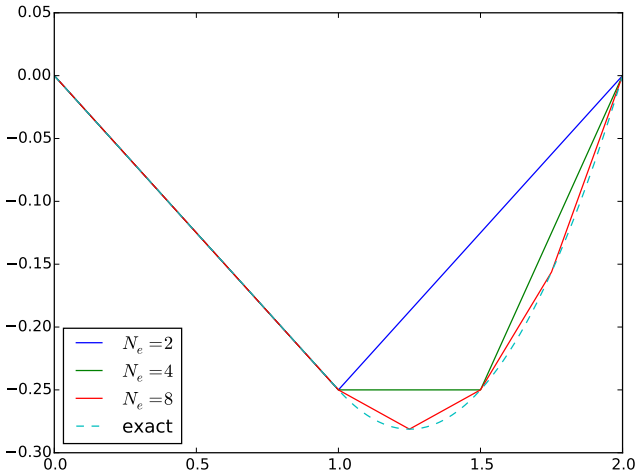
```
def plot_sine_solution():
    x = np.linspace(0, 2, 101)
    u = sine_solution(x, N=20)
    plt.figure()
    x = np.linspace(0, 2, 101)
    plt.plot(x, exact_solution(x), '--')
    N_values = 0, 1, 5
    for N in 0, 1, 5, 10:
        plt.plot(x, u[N])
    plt.legend(['exact'] + ['N=%d' % N for N in N_values])
    plt.savefig('tmp2.png');  plt.savefig('tmp2.pdf')
```



c) Solve the problem with P1 finite elements. Plot the solution for $N_e = 2, 4, 8$ elements.

Solution. The element matrices and vectors are as for the well-known model problem $u'' = 1$, except that the element vectors vanish for all elements in $[0, 1]$. The following function defines a uniform mesh of P1 elements and runs a finite element algorithm where we use ready-made/known formulas for the element matrix and vector:

```
def P1_solution():
    plt.figure()
    from fe1D import mesh_uniform, u_glob
    N_e_values = [2, 4, 8]
    d = 1
    legends = []
    for N_e in N_e_values:
        vertices, cells, dof_map = mesh_uniform(
            N_e=N_e, d=d, Omega=[0,2], symbolic=False)
        h = vertices[1] - vertices[0]
        Ae = 1./h*np.array(
            [[1, -1],
             [-1, 1]])
        N = N_e + 1
        A = np.zeros((N, N))
        b = np.zeros(N)
        for e in range(N_e):
            if vertices[e] >= 1:
                be = -h/2.*np.array(
                    [1, 1])
            else:
                be = h/2.*np.array(
                    [0, 0])
            for r in range(d+1):
                for s in range(d+1):
                    A[dof_map[e][r], dof_map[e][s]] += Ae[r,s]
                b[dof_map[e][r]] += be[r]
        # Enforce boundary conditions
        A[0,:] = 0; A[0,0] = 1; b[0] = 0
        A[-1,:] = 0; A[-1,-1] = 1; b[-1] = 0
        c = np.linalg.solve(A, b)

        # Plot solution
        xc, u, nodes = u_glob(c, vertices, cells, dof_map)
        plt.plot(xc, u)
        legends.append('$N_e=%d$' % N_e)
    plt.plot(xc, exact_solution(xc), '--')
    legends.append('exact')
    plt.legend(legends, loc='lower left')
    plt.savefig('tmp3.png'); plt.savefig('tmp3.pdf')
```



Filename: `cable_discont_load`.

Exercise 8: Compute with a non-uniform mesh

a) Derive the linear system for the problem $-u'' = 2$ on $[0, 1]$, with $u(0) = 0$ and $u(1) = 1$, using P1 elements and a *non-uniform* mesh. The vertices have coordinates $x_0 = 0 < x_1 < \cdots < x_{N_n-1} = 1$, and the length of cell number e is $h_e = x_{e+1} - x_e$.

Solution. The element matrix and vector for this problem is given by (61). The change in this exercise is that h is not a constant element length, but varying with the element number e. We therefore write

$$\tilde{A}^{(e)} = \frac{1}{h_e}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \quad \tilde{b}^{(e)} = h_e \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Assembling such element matrices yields

$$\begin{pmatrix}
h_0^{-1} & -h_0^{-1} & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
-h_0^{-1} & h_0^{-1}+h_1^{-1} & -h_1^{-1} & \ddots & & & & & \vdots \\
0 & -h_1^{-1} & h_1^{-1}+h_2^{-1} & -h_2^{-1} & \ddots & & & & \vdots \\
\vdots & \ddots & & \ddots & \ddots & 0 & & & \vdots \\
\vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\
\vdots & & & 0 & -h_{i-1}^{-1} & h_{i-1}^{-1}+h_i^{-1} & -h_i^{-1} & \ddots & \vdots \\
\vdots & & & & \ddots & \ddots & \ddots & \ddots & 0 \\
\vdots & & & & & \ddots & \ddots & \ddots & -h_{N_e}^{-1} \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & -h_{N_e}^{-1} & h_{N_e}^{-1}
\end{pmatrix}$$

The element vectors assemble to

$$\begin{pmatrix}
h_0 \\
h_0+h_1 \\
\vdots \\
\vdots \\
\vdots \\
h_{i-1}+h_i \\
\vdots \\
\vdots \\
h_{N_e}
\end{pmatrix}$$

b) It is of interest to compare the discrete equations for the finite element method in a non-uniform mesh with the corresponding discrete equations arising from a finite difference method. Go through the derivation of the finite difference formula $u''(x_i) \approx [D_x D_x u]_i$ and modify it to find a natural discretization of $u''(x_i)$ on a non-uniform mesh. Compare the finite element and difference discretizations

Solution. Using the definition of the centered, 2nd-order finite difference approximation to u'' we can set up

$$[D_x D_x u]_i = [D_x(D_x u)]_i = \frac{\frac{u_{i+1}-u_i}{x_{i+1}-x_i} - \frac{u_i-u_{i-1}}{x_i-x_{i-1}}}{x_{i+1/2}-x_{i-1/2}}\,.$$

Now,

$$x_{i+1/2}-x_{i-1/2} = \frac{1}{2}(x_i-x_{i-1}) + \frac{1}{2}(x_{i+1}-x_i) = \frac{1}{2}(x_{i+1}-x_{i-1})\,.$$

We then get the difference equation

$$u''(x_i) \approx \frac{2}{h_i + h_{i-1}} \left(\frac{u_{i+1} - u_i}{h_i} - \frac{u_i - u_{i-1}}{h_{i-1}} \right) = 2\,.$$

The factor 2 on either side cancels.

Looking at the finite element equations in a), the equation for a general row i reads

$$-\frac{1}{h_{i-1}} c_{i-1} + (\frac{1}{h_{i-1}} - \frac{1}{h_i}) c_i + \frac{1}{h_i} c_{i+1} = h_{i-1} + h_i\,.$$

Replacing c_i by u_i (assuming we keep unknowns at all nodes) and rearranging gives

$$\frac{1}{h_{i-1}}(u_i - u_{i-1}) - \frac{1}{h_i}(u_{i+1} - u_i) = h_{i-1} + h_i\,.$$

Dividing by the right-hand side gives

$$\frac{1}{h_{i-1} + h_i} \left(\frac{1}{h_{i-1}}(u_i - u_{i-1}) - \frac{1}{h_i}(u_{i+1} - u_i) \right) = 1\,.$$

This is the same difference equation as we have in the finite difference method.
Filename: `nonuniform_P1`.

Problem 9: Solve a 1D finite element problem by hand

The following scaled 1D problem is a very simple, yet relevant, model for convective transport in fluids:

$$u' = \epsilon u'', \quad u(0) = 0,\ u(1) = 1,\ x \in [0, 1]\,. \tag{96}$$

a) Find the analytical solution to this problem. (Introduce $w = u'$, solve the first-order differential equation for $w(x)$, and integrate once more.)

b) Derive the variational form of this problem.

c) Introduce a finite element mesh with uniform partitioning. Use P1 elements and compute the element matrix and vector for a general element.

d) Incorporate the boundary conditions and assemble the element contributions.

e) Identify the resulting linear system as a finite difference discretization of the differential equation using

$$[D_{2x} u = \epsilon D_x D_x u]_i\,.$$

f) Compute the numerical solution and plot it together with the exact solution for a mesh with 20 elements and $\epsilon = 10, 1, 0.1, 0.01$.
Filename: `convdiff1D_P1`.

Exercise 10: Investigate exact finite element solutions

Consider

$$-u''(x) = x^m, \quad x \in (0, L), \quad u'(0) = C,\ u(L) = D,$$

where $m \geq 0$ is an integer, and L, C, and D are given numbers. Utilize a mesh with two (non-uniform) elements: $\Omega^{(0)} = [0, 3]$ and $\Omega^{(0)} = [3, 4]$. Plot the exact solution and the finite element solution for $d = 1, 2, 3, 4$ and $m = 0, 1, 2, 3, 4$. Find values of d and m that make the finite element solution exact at the nodes in the mesh.

Hint. Use the `mesh_uniform`, `finite_element1D`, and `u_glob2` functions from the `fe1D.py` module.

Solution. The `model2` function from Section 1.2 can find the exact solution by `model2(x**m, L, C, D)`. We fix, for simplicity, the values of L, C, and D as $L = 4$, $C = 5$, and $D = 2$. After calculating a symbolic solution, we can convert the expression to a Python function with `sympy.lambdify`. For each d value we then create a uniform mesh and displace the vertex with number 1 to the value 3. The various functions for specifying the element matrix and vector entries are as given in Section 6.4, since the model problem is the same. Our code then becomes

```
from u_xx_f_sympy import model2, x
import sympy as sym
import numpy as np
from fe1D import finite_element1D, mesh_uniform, u_glob
import matplotlib.pyplot as plt

C = 5
D = 2
L = 4

m_values = [0, 1, 2, 3, 4]
d_values = [1, 2, 3, 4]
for m in m_values:
    u = model2(x**m, L, C, D)
    print '\nm=%d, u: %s' % (m, u)
    u_exact = sym.lambdify([x], u)

    for d in d_values:
        vertices, cells, dof_map = mesh_uniform(
            N_e=2, d=d, Omega=[0,L], symbolic=False)
        vertices[1] = 3  # displace vertex
        essbc = {}
        essbc[dof_map[-1][-1]] = D

        c, A, b, timing = finite_element1D(
            vertices, cells, dof_map,
            essbc,
            ilhs=lambda e, phi, r, s, X, x, h:
            phi[1][r](X, h)*phi[1][s](X, h),
            irhs=lambda e, phi, r, X, x, h:
            x**m*phi[0][r](X),
```

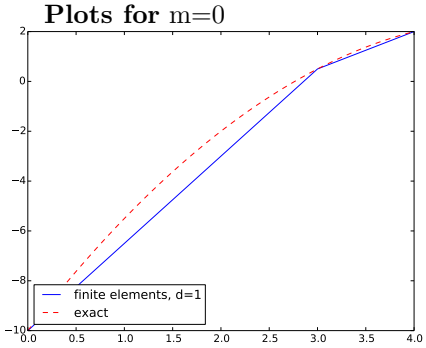
```
            blhs=lambda e, phi, r, s, X, x, h: 0,
            brhs=lambda e, phi, r, X, x, h:
            -C*phi[0][r](-1) if e == 0 else 0,
            intrule='GaussLegendre')

        # Visualize
        # (Recall that x is a symbol, use xc for coordinates)
        xc, u, nodes = u_glob(c, vertices, cells, dof_map)
        u_e = u_exact(xc)
        print 'Max diff at nodes, d=%d:' % d, \
              np.abs(u_exact(nodes) - c).max()
        plt.figure()
        plt.plot(xc, u, 'b-', xc, u_e, 'r--')
        plt.legend(['finite elements, d=%d' %d, 'exact'],
                   loc='lower left')
        figname = 'tmp_%d_%d' % (m, d)
        plt.savefig(figname + '.png'); plt.savefig(figname + '.pdf')
```

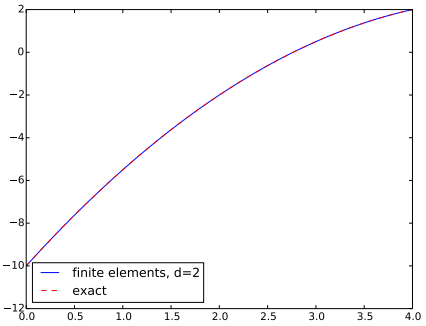
First we look at the numerical solution at the nodes:

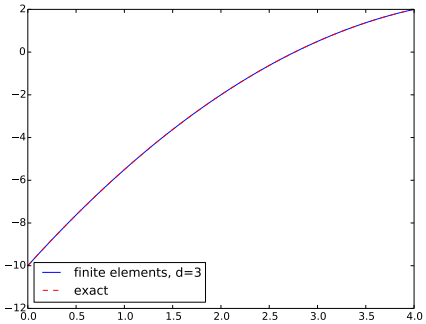
```
m=0, u: -x**2/2 + 5*x - 10
Max diff at nodes, d=1: 2.22044604925e-16
Max diff at nodes, d=2: 3.5527136788e-15
Max diff at nodes, d=3: 1.7763568394e-15
Max diff at nodes, d=4: 2.46913600677e-13

m=1, u: -x**3/6 + 5*x - 22/3
Max diff at nodes, d=1: 8.881784197e-16
Max diff at nodes, d=2: 1.7763568394e-15
Max diff at nodes, d=3: 7.9936057773e-15
Max diff at nodes, d=4: 3.01092484278e-13

m=2, u: -x**4/12 + 5*x + 10/3
Max diff at nodes, d=1: 3.10862446895e-15
Max diff at nodes, d=2: 0.084375
Max diff at nodes, d=3: 0.0333333333333
Max diff at nodes, d=4: 5.20472553944e-13

m=3, u: -x**5/20 + 5*x + 166/5
Max diff at nodes, d=1: 1.35555555556
Max diff at nodes, d=2: 0.3796875
Max diff at nodes, d=3: 0.185714285714
Max diff at nodes, d=4: 0.0254255022334

m=4, u: -x**6/30 + 5*x + 1778/15
Max diff at nodes, d=1: 4.8
Max diff at nodes, d=2: 1.4428125
Max diff at nodes, d=3: 0.719047619047
Max diff at nodes, d=4: 0.16865583147
```

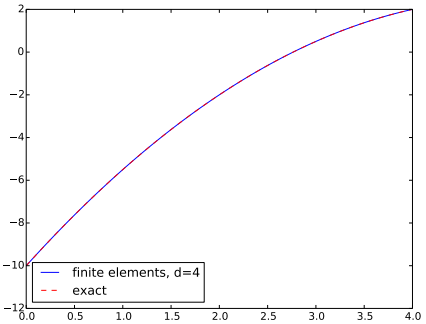
We observe that all elements are capable of computing the exact values at the nodes for $m = 0$ and $m = 1$. With $m = 0$, the solution is quadratic in x, and P2, P3, and P4 will be exact. It is more of a surprise that also the P1 elements are exact in this case. A peculiar feature is that P1 elements are also exact at the nodes $m = 2$, but not P2 and P3 elements (the solution goes like x^4 so it is not surprising that P2 and P3 elements give a numerical error also at the nodes). Clearly, P4 elements produce the exact solution for $m = 4$ since u is a polynomial of degree 4. For larger m values we have discrepancy between the numerical and exact values at the nodes.

Plots for m=0

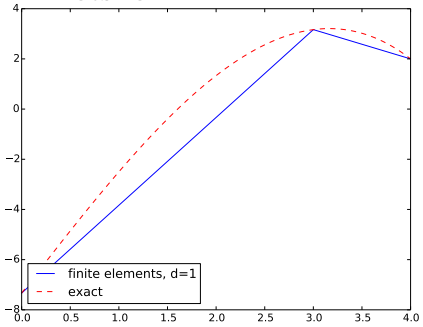


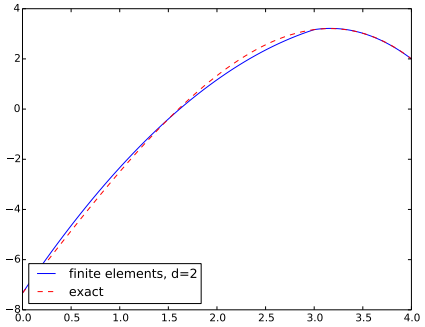


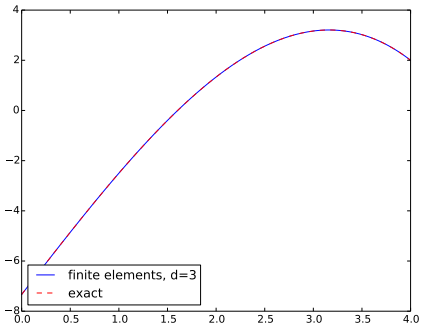


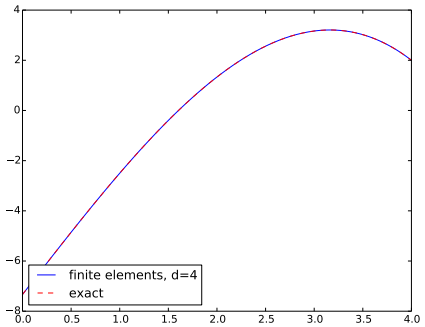


Plots for m=1

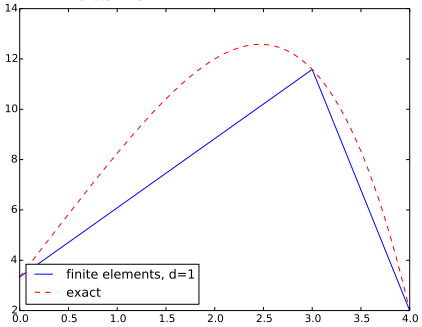


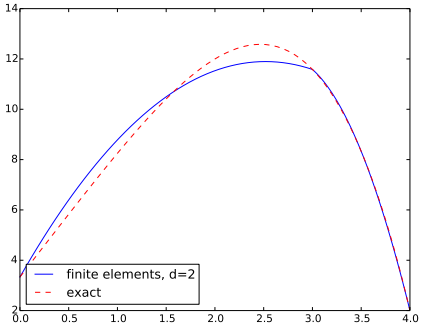


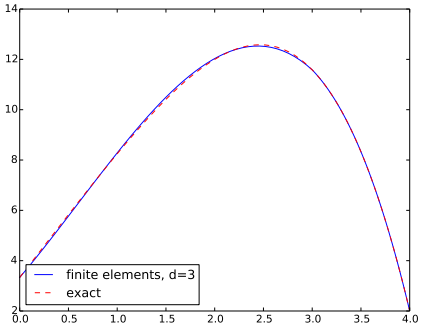


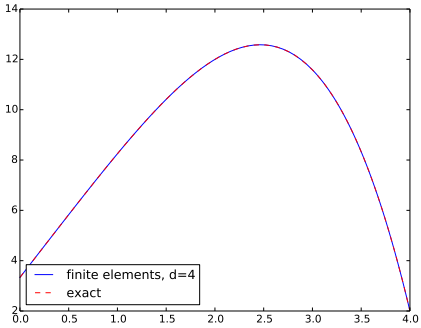


Plots for m=2

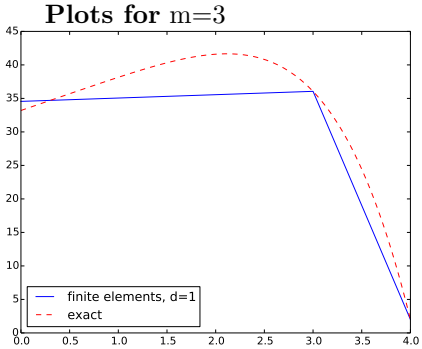


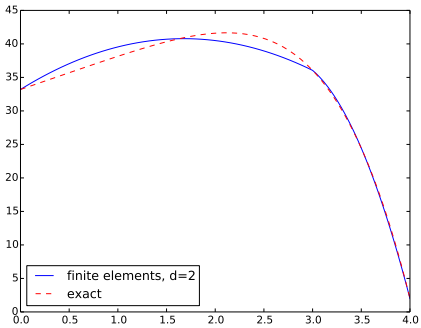


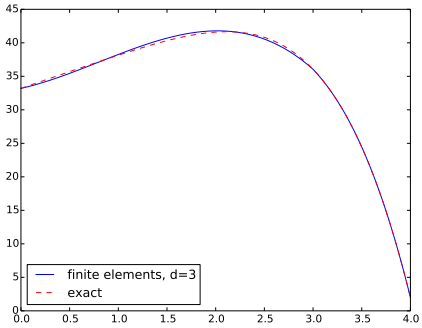


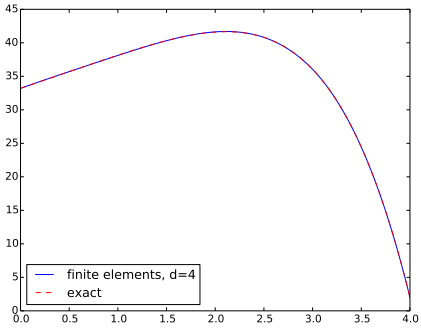


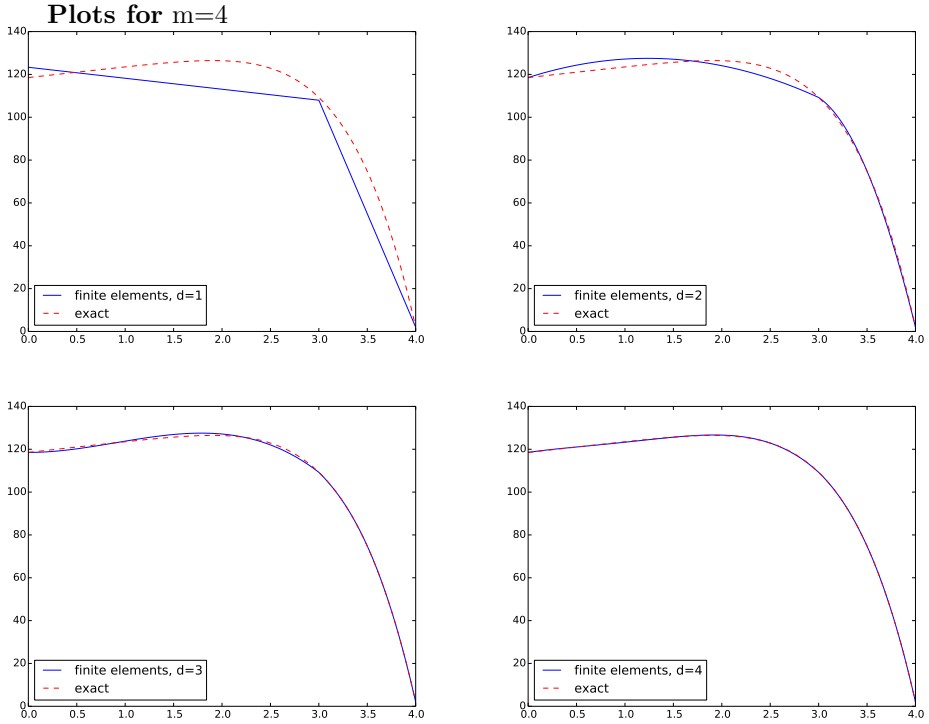
Plots for m=3











Filename: `u_xx_xm_P1to4`.

Exercise 11: Compare finite elements and differences for a radially symmetric Poisson equation

We consider the Poisson problem in a disk with radius R with Dirichlet conditions at the boundary. Given that the solution is radially symmetric and hence dependent only on the radial coordinate ($r = \sqrt{x^2 + y^2}$), we can reduce the problem to a 1D Poisson equation

$$-\frac{1}{r}\frac{d}{dr}\left(r\frac{du}{dr}\right) = f(r), \quad r \in (0, R),\ u'(0) = 0,\ u(R) = U_R\,. \tag{97}$$

a) Derive a variational form of (97) by integrating over the whole disk, or posed equivalently: use a weighting function $2\pi r v(r)$ and integrate r from 0 to R.

b) Use a uniform mesh partition with P1 elements and show what the resulting set of equations becomes. Integrate the matrix entries exact by hand, but use a Trapezoidal rule to integrate the f term.

c) Explain that an intuitive finite difference method applied to (97) gives

$$\frac{1}{r_i}\frac{1}{h^2}\left(r_{i+\frac{1}{2}}(u_{i+1} - u_i) - r_{i-\frac{1}{2}}(u_i - u_{i-1})\right) = f_i, \quad i = rh\,.$$

For $i = 0$ the factor $1/r_i$ seemingly becomes problematic. One must always have $u'(0) = 0$, because of the radial symmetry, which implies $u_{-1} = u_1$, if we allow introduction of a fictitious value u_{-1}. Using this u_{-1} in the difference equation for $i = 0$ gives

$$\frac{1}{r_0}\frac{1}{h^2}\left(r_{\frac{1}{2}}(u_1 - u_0) - r_{-\frac{1}{2}}(u_0 - u_1)\right) = \frac{1}{r_0}\frac{1}{2h^2}\left((r_0 + r_1)(u_1 - u_0) - (r_{-1} + r_0)(u_0 - u_1)\right) \approx 2(u_1 - u_0),$$

if we use $r_{-1} + r_1 \approx 2r_0$.

Set up the complete set of equations for the finite difference method and compare to the finite element method in case a Trapezoidal rule is used to integrate the f term in the latter method.
Filename: `radial_Poisson1D_P1`.

Exercise 12: Compute with variable coefficients and P1 elements by hand

Consider the problem

$$-\frac{d}{dx}\left(\alpha(x)\frac{du}{dx}\right) + \gamma u = f(x), \quad x \in \Omega = [0, L], \quad u(0) = \alpha,\ u'(L) = \beta\,. \tag{98}$$

We choose $\alpha(x) = 1 + x^2$. Then

$$u(x) = \alpha + \beta(1 + L^2)\tan^{-1}(x), \tag{99}$$

is an exact solution if $f(x) = \gamma u$.

Derive a variational formulation and compute general expressions for the element matrix and vector in an arbitrary element, using P1 elements and a uniform partitioning of $[0, L]$. The right-hand side integral is challenging and can be computed by a numerical integration rule. The Trapezoidal rule (109) gives particularly simple expressions. Filename: `atan1D_P1`.

Exercise 13: Solve a 2D Poisson equation using polynomials and sines

The classical problem of applying a torque to the ends of a rod can be modeled by a Poisson equation defined in the cross section Ω:

$$-\nabla^2 u = 2, \quad (x, y) \in \Omega,$$

with $u = 0$ on $\partial\Omega$. Exactly the same problem arises for the deflection of a membrane with shape Ω under a constant load.

For a circular cross section one can readily find an analytical solution. For a rectangular cross section the analytical approach ends up with a sine series. The

idea in this exercise is to use a single basis function to obtain an approximate answer.

We assume for simplicity that the cross section is the unit square: $\Omega = [0,1] \times [0,1]$.

a) We consider the basis $\psi_{p,q}(x,y) = \sin((p+1)\pi x)\sin(q\pi y)$, $p,q = 0,\ldots,n$. These basis functions fulfill the Dirichlet condition. Use a Galerkin method and $n = 0$.

b) The basis function involving sine functions are orthogonal. Use this property in the Galerkin method to derive the coefficients $c_{p,q}$ in a formula $u = \sum_p \sum_q c_{p,q}\psi_{p,q}(x,y)$.

c) Another possible basis is $\psi_i(x,y) = (x(1-x)y(1-y))^{i+1}$, $i = 0,\ldots,N$. Use the Galerkin method to compute the solution for $N = 0$. Which choice of a single basis function is best, $u \sim x(1-x)y(1-y)$ or $u \sim \sin(\pi x)\sin(\pi y)$? In order to answer the question, it is necessary to search the web or the literature for an accurate estimate of the maximum u value at $x = y = 1/2$.
Filename: `torsion_sin_xy`.

References

[1] D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, third edition, 2007.

[2] S. Brenner and R. Scott. *The Mathematical Theory of Finite Element Methods*. Springer, third edition, 2007.

[3] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, second edition, 1996.

[4] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Dover, 2009.

[5] H. P. Langtangen. Approximation of functions. `http://tinyurl.com/k3sdbuv/pub/approx`.

[6] M. G. Larson and F. Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Texts in Computational Science and Engineering. Springer, 2013.

[7] A. Quarteroni and A. Valli. *Numerical Approximation of Partial Differential Equations*. Springer, 1994.

Index